Introduction to Algorithms

Department of Computer Science and Engineering

East China University of Science and Technology

Lecture 05

Greedy Algorithm

The Greedy Algorithm

- A **greedy algorithm** is an algorithmic paradigm that follows the problem solving heuristic of making the locally optimal choice at each stage with the intent of finding a global optimum.
- In many problems, a greedy strategy **does not** usually produce an optimal solution, but nonetheless a greedy heuristic may yield locally optimal solutions that approximate a globally optimal solution in a reasonable amount of time.

Greedy algorithms

- Greedy algorithms have a long history of study in combinatorial optimization and theoretical computer science. Greedy heuristics are known to produce suboptimal results on many problems, and so natural questions are:
- For which problems do greedy algorithms perform optimally?
- For which problems do greedy algorithms guarantee an approximately optimal solution?
- For which problems is the greedy algorithm guaranteed *not* to produce an optimal solution?

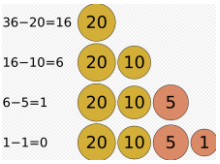
The Greedy Algorithm

- For example, a greedy strategy for the traveling salesman problem (which is of a high computational complexity) is the following heuristic:
- "At each step of the journey, visit the nearest unvisited city."
- This heuristic ***does not intend to*** find a best solution, but it terminates in a reasonable number of steps; finding an optimal solution to such a complex problem typically requires unreasonably many steps.
- In mathematical optimization, greedy algorithms optimally solve combinatorial problems having the properties of matroids, and give constant-factor approximations to optimization problems with submodular structure.

The Greedy Algorithm-a successful case

- Greedy algorithms determine minimum number of coins to give while making change. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 5, 10, 20}.

The coin of the highest value, less than the remaining change owed, is the local optimum.



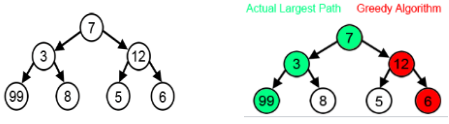
The Greedy Algorithm-a unsuccessful case

- Greedy algorithms determine minimum number of coins to give while making change. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using only coins with values {1, 8, 10, 20}.
- 36-20*1=16
- 16-10*1=6
- 6-1*6=0
- {1,1, 1,1,1,1,10, 20}.
- But an optimal solution:{8,8, 20}.

The Greedy Algorithm-a successful case

- (In general the change-making problem requires dynamic programming to find an optimal solution; however, most currency systems, including the Euro and US Dollar, are special cases where the greedy strategy does find an optimal solution.)

The Greedy Algorithm-a failture case



With a goal of reaching the largest-sum, at each step, the greedy algorithm will choose what appears to be the optimal immediate choice, so it will choose 12 instead of 3 at the second step, and will not reach the best solution, which contains 99.

The Greedy Algorithm

- In general, greedy algorithms have five components:

① A candidate set, from which a solution is created

② A selection function, which chooses the best candidate to be added to the solution

③ A feasibility function, that is used to determine if a candidate can be used to contribute to a solution

④ An objective function, which assigns a value to a solution, or a partial solution, and

⑤ A solution function, which will indicate when we have discovered a complete solution

the Greedy Algorithm

- For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the *unique worst possible* solution.
- One example is the traveling salesman problem mentioned above: for each number of cities, there is an assignment of distances between the cities for which the nearest-neighbor heuristic produces the unique worst possible tour.

- Greedy algorithms don't always yield an optimal solution.
- But sometimes they do. We'll see a problem for which they do.
- Then we'll look at some general characteristics of when greedy algorithms give optimal solutions.

two properties of the Greedy Algorithm

- Greedy algorithms produce good solutions on some mathematical problems, but not on others.
- Most problems for which they work will have two properties:

① **Greedy choice property**

② **Optimal substructure**

Greedy choice property

- **Greedy choice property**
- We can make whatever choice seems best at the moment and then solve the subproblems that arise later.
- The choice made by a greedy algorithm may depend on choices made so far, but not on future choices or all the solutions to the subproblem.
- It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices.
- This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Optimal substructure

- **Optimal substructure**
- "A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems.
- choice property

- **Scheduling**
  - **Activity Selection**
- **Graph Algorithms**
  - **Minimum Spanning Trees**
  - **Dijkstra's (shortest path) Algorithm**
  - **Bellman-Ford Algorithm**
  - **Floyd-Warshall Algorithm**
- **Other Heuristics**
  - **Huffman coding**

Greedy Algorithm

- **Similar to dynamic programming**
- **Used for optimization problems**

Greedy Algorithm

- **IDEA.** A *greedy algorithm* always makes the choice that looks best at the moment
  - **The hope:** a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
    - Yes: fractional knapsack problem
    - No: 0-1 knapsack problem

Example 1. Activity-Selection Problem

Activity-Selection Problem

- Several competing activities that require exclusive use of common resources (Same place at a fixed period of time), with the goal is of selecting a maximum-size set of mutually compatible activities that wish to use a resource (a period of time in the p l a c e)

Activity-Selection Problem

- Set of activities $S = \{a_1, a_2, \ldots, a_n\}$.
- Each activity a_i has a **start time s_i** and **finish time f_i**, where $0 \le s_i < f_i < \infty$.
  - Goal: Select a **maximum-size** subset of S contains mutually compatible activities.
  - Note: Could have many other objectives:
    - Schedule room for longest time.
    - Maximize income rental fees.

- **Compatible【相容】activities**: We call two activities are compatible if the two intervals does not overlap
- (i.e., a_i, a_j are compatible if $s_i \ge f_j$ or $s_j \ge f_i$)

- **Example: *S sorted by finish time:***

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

i	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

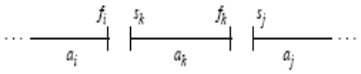
- **最大的相容集是**
- **$\{a_1, a_3, a_6, a_8\}$**
- **$\{a_2, a_5, a_7, a_9\}$**

Definition of Solution

- A subset A of S is a solution if all the activities in A are mutually compatible.
- We call a solution A is optimal if it is a solution whose size is one of the longest.

Step1:Optimal Substructure of Activity Selection

- Let $S_{ij}=\{a_k \text{ in } S: f_i \le s_k < f_k \le s_j\}$
- then S_{ij} is the subset of S that can start after activity a_i finishes and finish before activity a_j starts



原问题是什么？

- Adding two fictitious activities a_0 and a_{n+1} such that $f_0=0$ and $s_{n+1} = \infty$, then $S=S_{0,n+1}$
- 原问题$S=S_{0,n+1}$!

Assumption

- Assumption: the activities are sorted by its finish time:

$$f_0 \le f_1 \le f_2 \le \ldots \le f_n < f_{n+1}$$

Properties

- $S_{ij}=0$ whenever $i \ge j$
- – **If there exists $a_k \in S_{ij}$:**
- $f_i \le s_k < f_k \le s_j < f_j \Rightarrow f_i < f_j$.
- – **But $i \ge j \Rightarrow f_i \ge f_j$, *a contradiction***

- If we sorted the activities in monotonically increasing order of finish time, then the subproblem space is a subset of the $\{S_{ij}\}$ for all $0 \le i < j \le n+1$.

Substructure

- **Suppose that a solution to S_{ij} *includes* a_m. *Have 2 subproblems:***
  - S_{im} *(start after* a_i *finishes, finish before* a_m *starts)*
  - S_{mj} *(start after* a_m *finishes, finish before* a_j *starts)*

- Solution to S_{ij} is (solution to S_{im}) $\cup \{a_m\} \cup$ (solution to S_{mj}).
- *|Solution to S| = |solution to* S_{im}*| + 1 + |solution to* S_{mj} *|*

Optimal substructure

- **Let** A_{ij}**=** ***optimal solution to*** S_{ij}.
- Suppose A_{ij} is an optimal solution of problem, and a_m is in A_{ij}, then the solutions A_{im} to S_{im} and A_{mj} to S_{mj} used within this optimal solution to S_{ij} must be optimal as well.
- **"cut-and-paste"** method:

  If a solution A'_{im} to S_{im} contians more elements than A_{im}, we could cut out A_{im} from A_{ij} and paste in A'_{im}, thus producing an another solution to S_{ij} which has more activities than A_{ij}, which produces a contradiction!

Optimal solution can be constructed by optimal solutions of subproblems

- If a_m is contains in an optimal solution of S_{ij} than any optimal solution A_{ij} which contains a_m can be expressed by

$$A_{ij} = A_{im} \cup \{a_m\} \cup A_{mj}$$

- Where A_{im} and A_{mj} are optimal solutions for S_{im} and S_{mj}
- Overlap subproblems?

Step2: A recursive solution

- Let c[i, j] be the number of the activities in an optimal solution to S_{ij}, we have c[i, j] = 0 whenever S_{ij} = Φ;

$$c[i,j] = \begin{cases} 0, & \text{if } S_{ij} = \phi \\ \max\{c[i,k] + c[k,j] + 1\}, & \text{if } S_{ij} \neq \phi \\ i<k<j, \\ a_k \in S_{ij} \end{cases}$$

◆k的取值为i＋1，…，j－1

Greedy-choice property

Theorem

Consider any nonempty subproblem S_{ij}, and let a_m be the activity in S_{ij} with the earliest finish time: f_m = min{f_k: a_k in S_{ij}}

Then

- Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}
- The subproblem S_{im} is empty

Proof of the theorem

Proof :

The second part of the theorem is trivial.

To prove the first part, let A_{ij} is an optimal solution of S_{ij}, and order the activities in A_{ij} in monotonically increasing order of finish time.

Let a_k be the first activity in A_{ij},

if a_k=a_m, then we have done

else we can replace a_k with a_m in A_{ij}, and it changes to another solution! It is optimal too!

Why theorem is valuable?

	before theorem	after theorem
# of subproblems in optimal solution	2	1
# of choices to consider	$j-i-1$	1

- We can compute the optimal solution in a topdown fashion, rather than the bottom-up manner used in dynamic programming.

The solution constructing procedure

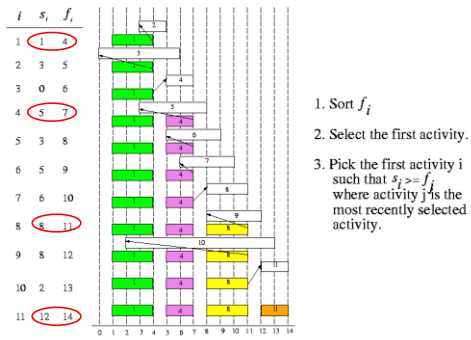
- Now we can solve top down:
  To solve a problem S_{ij},
  - Choose $a_m \in S_{ij}$ *with earliest finish time: the greedy choice.*
  - Then solve S_{mj}.

- What are the subproblems?
  - Original problem is $S_{0,n+1}$.
  - Suppose our first choice is a_{m1}.
  - Then next subproblem is $S_{m1,n+1}$.
  - Then one of the optimal solution is $\{a_{m1}\} \cup A_{m1,n+1}$.
    Where $A_{m1,n+1}$ is the optimal solution of $S_{m1,n+1}$
  - Suppose next choice is a_{m2}.
  - Next subproblem is $S_{m2,n+1}$.
  - And so on.

A Greedy Algorithm

- A **greedy algorithm** always makes the choice that looks best at the moment.

Activity Selection



1. Sort f_i
2. Select the first activity.
3. Pick the first activity i such that $s_i >= f_j$ where activity j is the most recently selected activity.

The Activity-Selection Algorithm

```
Greedy-Activity-Selector(s,f)
/* Assume f1 ≤ f2 ≤ ... ≤ fn. */
1. n ← length[s];
2. A ← {a1};
3. j ← 1;
4. for m ← 2 to n
5.     if sm ≥ fj
6.         A ← A ∪ { am };
7.         j ← m;
8. return A.
```

.Time complexity excluding sorting: .$O(n)$

Greedy Strategy

- The choice that seems best at the moment is the one we go with.
- What did we do for activity selection?
  - Determine the optimal substructure.
  - Develop a recursive solution.
  - Prove that at any stage of recursion, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice.
  - Show that all but one of the subproblems resulting from the greedy choice are empty.
  - Develop an iterative algorithm.

Greedy Strategy

- **Develop the substructure with an eye toward**
  - Making the greedy choice.
  - Leaving just one subproblem.

Elements of the Greedy Strategy

- When to apply greedy algorithms?
  - **Greedy-choice property:** A global optimal solution can be arrived at by making a locally optimal (greedy) choice.
    - Dynamic programming needs to check the solutions to subproblems.
  - **Optimal substructure:** An optimal solution to the problem contains within its optimal solutions to subproblems.
    - E.g., if A is an optimal solution to S, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S: s_i \geq f_1\}$.
- Greedy algorithms (*heuristics*) do not always produce optimal solutions.

Greedy vs Dynamic Programming

- Dynamic programming:
  - Make a choice at each step.
  - Choice depends on knowing optimal solutions to subproblems. Solve subproblems first.
  - Solve bottom-up.
- Greedy:
  - Make a choice at each step.
  - Make the choice before solving the subproblems.
  - Solve top-down.

Greedy vs Dynamic Programming

- Greedy algorithms v.s. dynamic programming (DP)
  - Common: optimal substructure
  - Difference: greedy-choice property
  - DP can be used if greedy solutions are not optimal.
- Examples

  0-1 knapsack problem

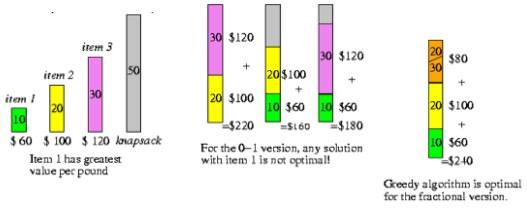
  Fractional knapsack problem

Knapsack Problem:

- Given n items, with ith item worth v_i dollars and weighing w_i pounds, a thief wants to take as valuable a load as possible, but can carry at most W pounds in his knapsack.
- **The 0-1 knapsack problem:** Each item is either taken or not taken (0-1 decision).
- **The fractional knapsack problem:** Allow to take fraction of items.

Exp:

- **Exp:** w = (10, 20, 30), v = (60, 100, 120), W = 50



Example2: Minimum spanning trees

Review Graph

- A graph G = (V, E)
  - V = set of vertices
  - E = set of edges = subset of V × V
  - Thus |E| = O(|V|²)

Graph Variations

- **连通图**(*connected graph)*
- a path from every vertex to every other
- **无向图***undirected graph:*
  - Edge (u,v) = edge (v,u)
  - No self-loops
- **有向图** *directed* graph:
  - Edge (u,v) goes from vertex u to vertex v, notated u→v

Graph Variations

- **带权图** *weighted graph*
- associates weights with either the edges or the vertices
  - E.g., a road map: edges might be weighted w/ distance
- **多重图** *multigraph*
- allows multiple edges between the same vertices
  - E.g., the call graph in a program (a function can get called from multiple points in another function)

Graph

- We will typically express running times in terms of |E| and |V| (often dropping the |'s)
  - If |E| ≈ |V|² 稠密
  - If |E| ≈ |V| 稀疏

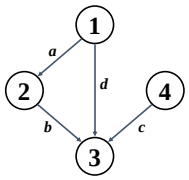
选择合适数据结构表示

Representing Graphs

- Assume V = {1, 2, …, *n*}
- *邻接矩阵adjacency matrix* :
  - A[*i*, *j*] = 1 if edge (*i*, *j*) ∈ E (or weight of edge)
    = 0 if edge (*i*, *j*) ∉ E

1. 邻接矩阵Adjacency Matrix

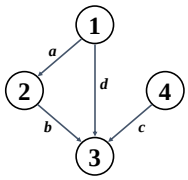
- Example:



A	1	2	3	4
1				
2				
3			??	
4				

Adjacency Matrix

- Example:



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

Adjacency Matrix

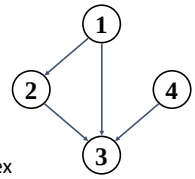
- *How much storage does the adjacency matrix require?*
- A: $O(V^2)$
  - A dense representation
- But, can be very efficient for small graphs
  - Especially if store just one bit/edge
  - Undirected graph: only need one diagonal of matrix

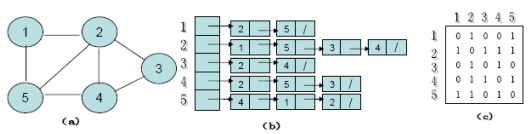
Adjacency Matrix

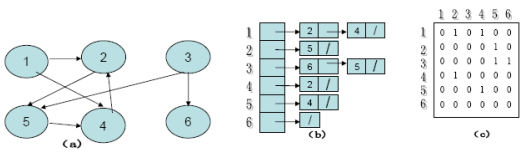
- The adjacency matrix is a dense representation
  - Usually too much storage for large graphs
  - But can be very efficient for small graphs
- Most large interesting graphs are sparse
  - E.g., planar graphs, in which no edges cross, have $|E| = O(|V|)$ by Euler's formula
  - For this reason the *adjacency list* is often a more appropriate respresentation

2. 邻接链表Adjacency List

- Adjacency list: for each vertex $v \in V$, store a list of vertices adjacent to v
- Example:
  - Adj[1] = {2,3}
  - Adj[2] = {3}
  - Adj[3] = {}
  - Adj[4] = {3}
- Variation: can also keep a list of edges coming *into* vertex





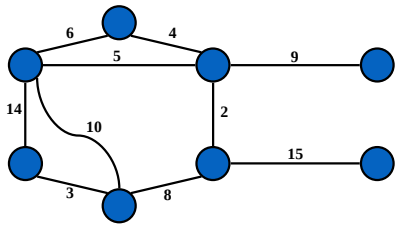


Adjacency List

- How much storage is required?
  - For directed graphs, # of items in adjacency lists is
    Σ out-degree(*v*) = |E|
    takes Θ(V + E) storage
  - For undirected graphs, # items in adj lists is
    Σ degree(v) = 2 |E| (*handshaking lemma*)
    also Θ(V + E) storage
- So: Adjacency lists take O(V+E) storage

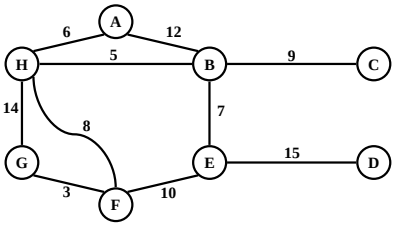
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph, find a *spanning tree* using edges that minimize the total weight



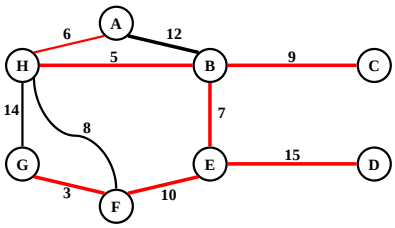
Minimum Spanning Tree

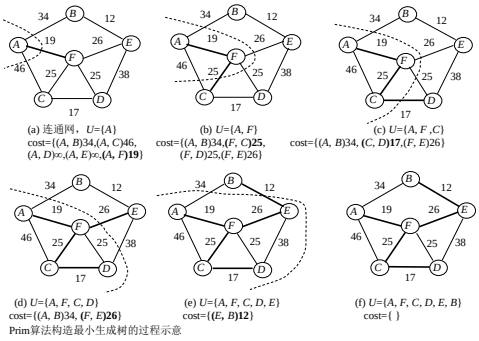
- *Which edges form the minimum spanning tree (MST) of the below graph?*

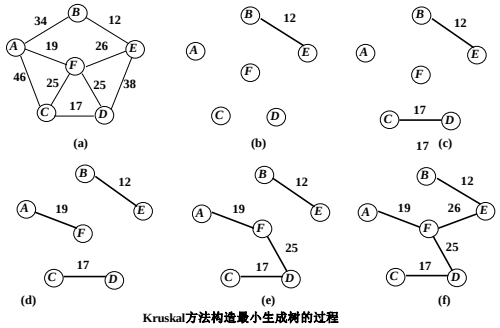


Minimum Spanning Tree

- Answer:







Kruskal方法构造最小生成树的过程

Minimum Spanning Tree

- *考虑:*
- *optimal substructure* property?

❖ overlapping subproblem?

Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property: an optimal tree is composed of optimal subtrees
  - Let T be an MST of G with an edge (u,v) in the middle
  - Removing (u,v) partitions T into two trees T_1 and T_2
  - Claim: T_1 is an MST of $G_1 = (V_1,E_1)$, and T_2 is an MST of $G_2 = (V_2,E_2)$
  - Proof: $w(T) = w(u,v) + w(T_1) + w(T_2)$
    (There can't be a better tree than T_1 or T_2, or T would be suboptimal)

Minimum Spanning Tree

- Thm:
  Let T be MST of G, and let $A \subseteq T$ be subtree of T
  Let (u,v) be min-weight edge connecting A to V-A
  Then $(u,v) \in T$

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What is the hidden cost in this code?

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

How often is ExtractMin() called?
How often is DecreaseKey() called?

Review: Prim's Algorithm

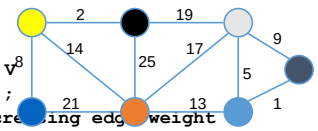
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What will be the running time?
A: Depends on queue

Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

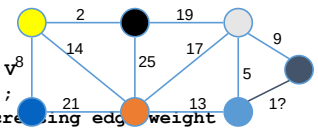
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

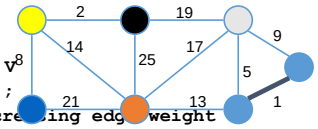
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

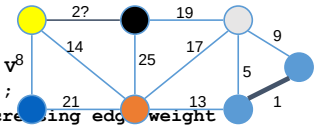
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
    T = ∅;
    for each v ∈ V
        MakeSet(v);
    sort E by increasing edge weight
    for each (u,v) ∈ E (in sorted order)
        if FindSet(u) ≠ FindSet(v)
            T = T ∪ {{u,v}};
            Union(FindSet(u), FindSet(v));
}
```

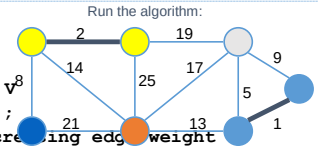
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

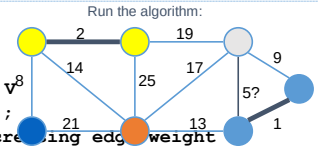
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

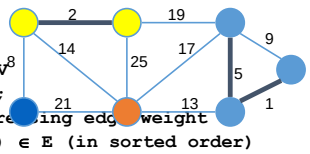
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

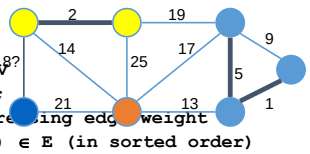
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

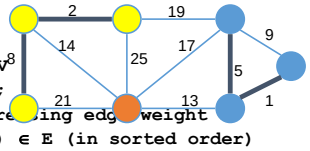
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

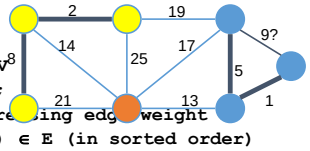
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

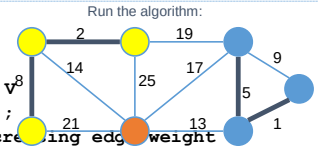
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

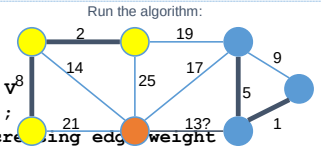
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

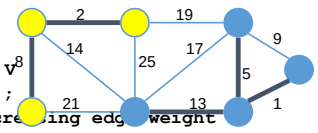
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

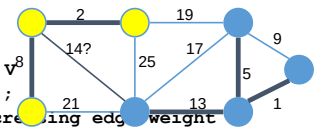
Run the algorithm:



Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

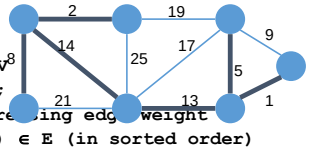
Run the algorithm:

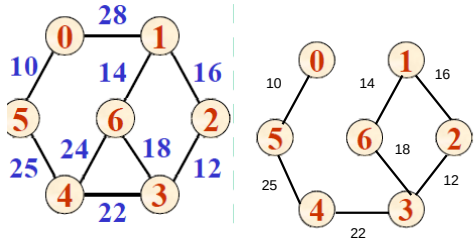


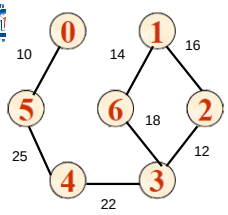
Kruskal's Algorithm

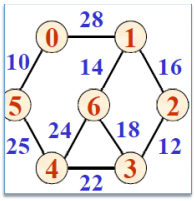
```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Run the algorithm:







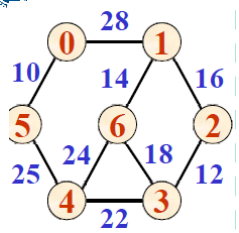


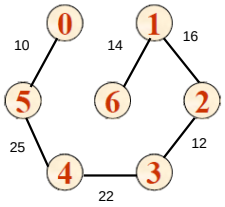
顶点集合状态：

{0,5} {1,6,2,3} {2,3}

{3} {4} {5} {6}

最小生成树的边的集合：





Example 3. **Huffman codes**

Coding

- Used for data compression, instruction-set encoding, etc.
- **Binary character code:** character is represented by a unique binary string
  - **Fixed-length code (block code):** *a*: 000, *b*: 001, ..., *f*: 101 ⇒ ace ↔ 000 010 100.
  - **Variable-length code:** frequent characters ⇒ short codeword; infrequent characters ⇒ long codeword

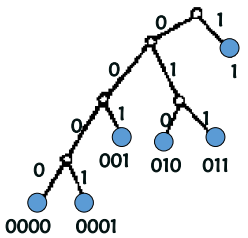
	a	b	c	d	e	f	cost / 100 characters
Frequency	45	13	12	16	9	5	
Fixed–length codeword	000	001	010	011	100	101	300
Variable–length codeword	0	101	100	111	1101	1100	224

Prefix Code

- **Prefix code:** No code is a prefix of some other code.

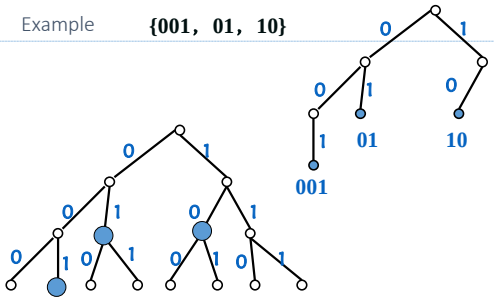
Binary Tree v.s. Prefix Code

Example



{0000, 0001, 001, 010, 011, 1}

Example {001, 01, 10}



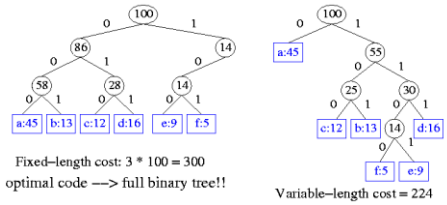
Optimal Prefix Code Design

- **Coding Cost** of T

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

  - c : each character in the alphabet C
  - $f(c)$: frequency of c
  - $d_T(c)$: depth of c's leaf (length of the codeword of c)

- **Code design:** Given $f(c_1), f(c_2), \ldots, f(c_n)$, construct a binary tree with n leaves such that $B(T)$ is minimized.
  - **Idea: more frequently used characters use shorter depth.**

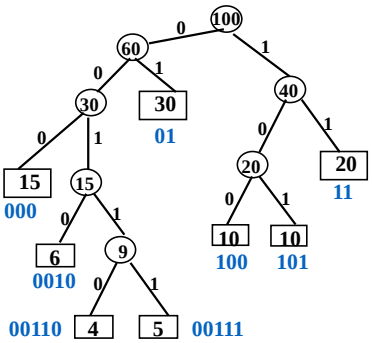


Huffman's Procedure

- **Pair** two nodes with the least costs at each step.

以频率作为权值，得

4， 5， 6， 10， 10， 15， 20， 30



Huffman's Algorithm

- Huffman(C)
- 1. $n \leftarrow |C|$;
- 2. $Q \leftarrow C$;
- 3. **for** $i \leftarrow 1$ **to** n-1
- 4. $z \leftarrow$ Allocate-Node();
- 5. $x \leftarrow left[z] \leftarrow$ Extract-Min(Q);
- 6. $y \leftarrow right[z] \leftarrow$ Extract-Min(Q);
- 7. $f[z] \leftarrow f[x]+f[y]$;
- 8. Insert(Q, z);
- 9. **return** Extract-Min(Q)

Time complexity: $O(n \lg n)$.
Extract-Min(Q) needs $O(\lg n)$ by a heap operation.
Requires initially $O(n \lg n)$ time to build a binary heap.

Huffman Algorithm:Optimal substucture

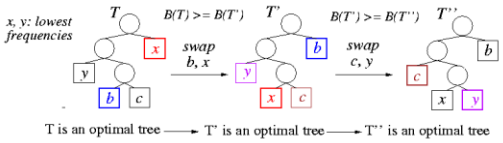
- **Optimal substructure:** Let *T* be a full binary tree for an optimal prefix code over *C*. Let *z* be the parent of two leaf characters *x* and *y*. If $f[z]=f[x]+f[y]$, tree $T' = T - \{x, y\}$ represents an optimal prefix code for $C' = C - \{x, y\} \cup \{z\}$.

$B(T) = B(T') + f(x) + f(y)$

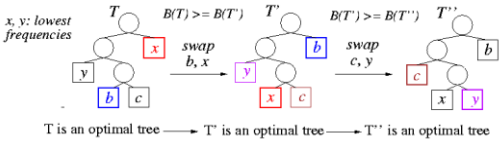
If T' is not optimal, find T'' s.t. B(T'') < B(T').
z in C' => z is a leaf of T'',
Add x, y as z's children (T''')
$\Rightarrow B(T''') = B(T'') + f(x) + f(y)$
$< B(T') + f(x) + f(y)$
$= B(T)$ *a contradiction!!*

Huffman Algorithm: Greedy Choice

- **Greedy choice:** Two characters *x* and *y* with the lowest frequencies must have *the same length* and differ only in the last bit.



不同树的成本之差



$$B(T) - B(T') = \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c)$$
$$= f(x) d_T(x) + f(b) d_T(b) - f(x) d_{T'}(x) - f(b) d_{T'}(b)$$
$$= f(x) d_T(x) + f(b) d_T(b) - f(x) d_T(b) - f(b) d_T(x)$$
$$= (f(b) - f(x))(d_T(b) - d_T(x)) \geq 0$$

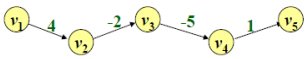
Example4: Single-Source Shortest Path

Paths in graphs

- Consider a digraph *G=(V, E), with edge-weight function w:E →R. The weight of path p = $v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is defined to* be

$$w(p) = \Sigma_{i=1..k-1} w(v_i, v_{i+1})$$

- **Example:**
- ***w(p) = -2***



Shortest paths

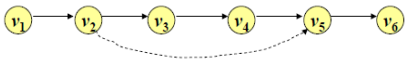
- A shortest path from *u to v is a path of minimum weight* from *u to v. The shortest-path weight from u to v is* defined as

min{*w(p): if there is a path p from u to v*}

- $\delta(u, v) =$ $\begin{cases} \infty \text{ otherwise} \end{cases}$

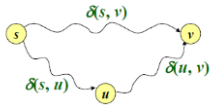
Optimal structure

- **Theorem.** A subgraph of a shortest path is a shortest path.
- Proof. Cut and paste:



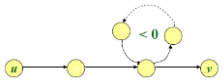
Triangle inequality

- **Theorem. For all *s, u, v* $\in$ *V, we have***
  $\delta(s, v) \le \delta(s, u) + \delta(u, v)$
- **Proof.**



Well-definedness of shortest paths

- If a graph *G contains a negative-weight cycle, then some* shortest paths may not exist.



Single-source shortest paths

- **Problem.** From a given source vertex $s \in V$, *find the* shortest-path weight $\delta(s, v)$ *for all* $v \in V$.
- If all edge weight *w(u, v) are nonnegative, all shortest-path* weights must exist.

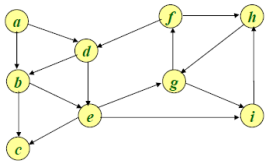
- **IDEA: Greedy**

1. Maintain a set *S of vertices whose shortest-path* distances from *s are known.*
2. At each step add to *S the vertex* $v \in V - S$ *whose* distance estimate from *s is minimal.*
3. Update the distance estimates of vertices adjacent to *v.*

Unweighted graphs

- Suppose that *w(u, v) = 1 for all* $(u, v) \in E$. *Can Dijkstra's* algorithm be improved?
- **Breadth-first search**

- Exercise: s=a



- (simplifying the second term in this replacement as necessary). For instance:
- 7/8=1/2+1/3+1/24。

Problem1-Egyptian fraction

- An Egyptian fraction is a representation of an irreducible fraction as a sum of distinct unit fractions, as e.g. 5/6 = 1/2 + 1/3.
- As the name indicates, these representations have been used as long ago as ancient Egypt, but the first published systematic method for constructing such expansions is described in the Liber Abaci (1202) of Leonardo of Pisa (Fibonacci).
- It is called a greedy algorithm because at each step the algorithm chooses greedily the largest possible unit fraction that can be used in any representation of the remaining fraction.
- For instance:
- **7/8=1/2+1/3+1/24。**