CSIEB0120
Lecture 04
Greedy Approach

Shiow-yang Wu & Wei-Che Chien
(吳秀陽 & 簡暐哲)
Department of Computer Science and Information Engineering
National Dong Hwa University

Objectives

- Describe the greedy approach
- Contrast the greedy and other approaches
- Solve optimization problems using the greedy approach
- Prove or disprove if greedy algorithm produces optimal solution
- When to use/avoid greedy algorithms

Greedy may not be that bad at all

- The word "greedy" seems to be always associated with bad things.
- In algorithm design, however, taking a greedy approach turns out to be quite intuitive and effective in certain cases.
- The basic idea is to tackle a difficult problem one step at a time based on the current status.
- The next step we choose is always to select the one that is expected to offer the most profit at current status. (thus the name greedy)

The Greedy Approach

- Unlike D&C or DP, we do not divide a problem into smaller instances.
- The approach is to take a sequence of steps that gradually leads us to the solution.
- Each step is determined by choosing the one that appears to be the best choice at that status. (i.e. locally optimal choice)
- Once a choice is made, it cannot be reconsidered.
- Choice made without regard to past or future choices. (We are just greedy!)

The Greedy Approach (contd.)

- The goal remains to be a globally optimal solution.
- Since each step is a locally optimal choice, greedy approach may not guarantee globally optimal solution.
- Global optimality must be proven.

The Greedy Way

- The problem is solved step-by-step iteratively.
- Initially the solution set is empty.
- At each iteration, items are added (usually one at a time) to the solution set.
- The items are selected in a greedy fasion. (more about this later)
- The process repeats until the solution set represents a real solution to the problem instance.
- More specifically, a greedy algorithm is designed with a set of components to help us carrying out the greedy way. (next slide)

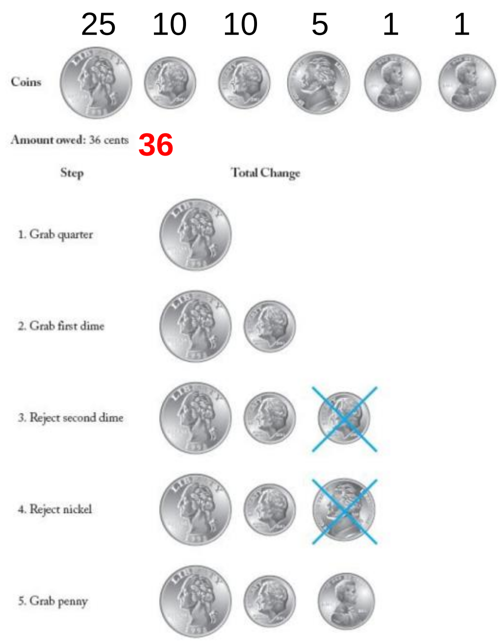
The Greedy Algorithm

- ***Selection procedure***: Choose the next item to add to the solution set according to the greedy criterion satisfying the locally optimal consideration.
- ***Feasibility Check***: Determine if the new set is feasible by determining if it is possible to complete this set to provide a solution to the problem instance.
- ***Solution Check:*** Determine whether the new set produced is a solution to the problem instance.

Example: Coin Change Problem

- A good example to demonstrate the greedy approach is the coin change problem.
- **Problem**: Given a set of coins and the change of a purchase, determine the set of coins that correctly make the change.
- The optimization version requires a solution set with minimum number of coins.
- The way most of us would take to solve this problem is exactly the greedy algorithm. (next slide)

Greedy Strategy for Coin Change Problem



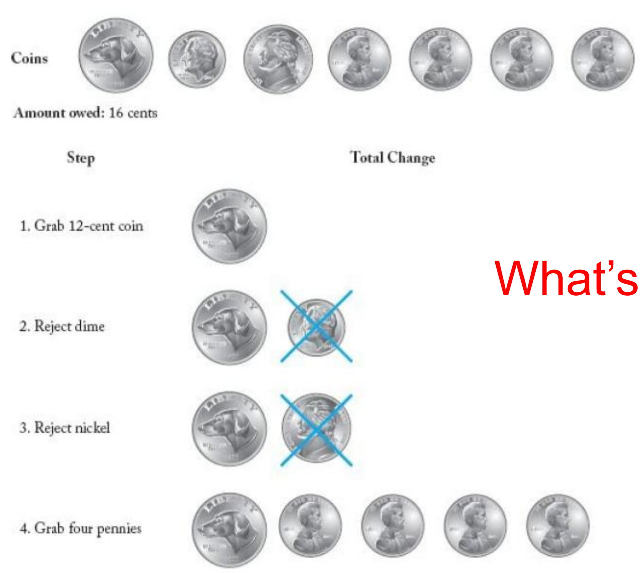
Greedy Algorithm for Coin Change

```
while (there are more coins and
       the instant is not solved) {
  grab the largest remaining coin; // selection
proc
  If (adding the coin makes the change
      exceed the amount owed)   // feasibility
check
     reject the coin;
  else
     add the coin to the change;
  If (the total value equals the amount owed)
                               // solution check
     the instance is solved;
}
```

Is the greedy solution optimal?

- If the set of coins is finite: {H, Q, D, N, P}
- With brute force, we can show that the greedy algorithm produces an optimal solution for the amount of $.01 - $.50
- Any amount of change > $.50 would be a multiple of what was shown (use induction)
- Include a 12-cent coin: i.e. { .50, .25, .12, .10, .05, .01 }
  - Produce $.16 in change would not be optimal !

Example of Non-optimal Solution



What's wrong?

Exercises

- Greedy approach itself never guarantees optimal solution.
- Thus, it is the responsibility of an algorithm designer to demonstrate that.
- OPTIMALITY PROOF is a MUST in greedy algorithm design.
- Exercises: Prove that the greedy algorithm always produces the minimum number of coins for a change if the coin set is {1, 5, 10}.

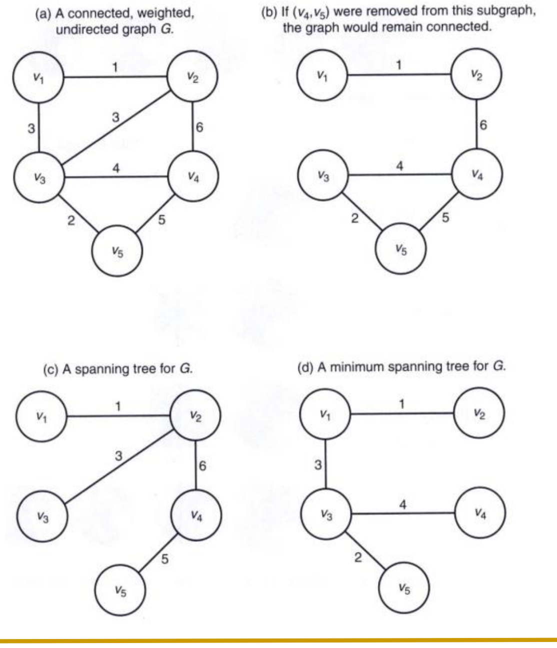
Minimum Spanning Tree Problem

- Graph theory terms:
  - Undirected graph G: G = (V, E)
  - Connected graph
  - Weighted graph
  - Path
  - Cyclic graph and acyclic graph
- Subgraph G' = (V', E') if V' (E') is a subset of V (E)
- Tree is an acyclic and connected graph.
- Spanning tree for G is an acyclic connected subgraph that contains ALL the nodes in G.
- Minimum spanning tree for G is a spanning tree of minimum weight (sum) for the graph.

MST Problem Definition

- Let G = (V , E)
- Let T be a spanning tree for G: T = (V, F) where $F \subseteq E$
- Find T such that the sum of the weights of the edges in F is minimal.
- **Applications**:
  - Interstate highway construction
  - Subway, railroad construction
  - Telecommunication network construction
  - Plumbing, wiring, power transmission, and etc.

Example of MST



Greedy Algorithms for Finding a Minimum Spanning Tree

- Prim's Algorithm
- Kruskal's Algorithm
- Each uses a different locally optimal property
- Must prove the optimality of each algorithm.

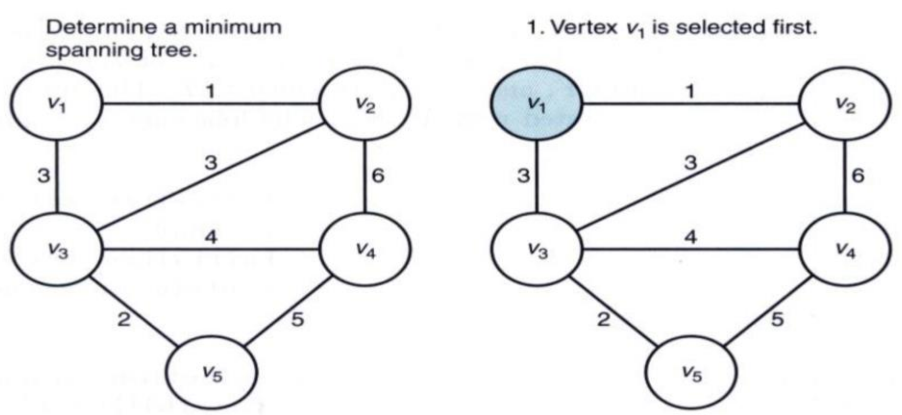
Prim's Algorithm

- Subset of edges F, initially empty.
- Subset of vertices Y initialized to an arbitrary vertex.
- $Y = \{v_1\}$
- Select a vertex nearest to Y from V-Y connected to a vertex in Y by a minimum weight edge
  - Add the selected vertex to Y
  - Add the edge connecting the selected vertex to F
- Ties broken arbitrarily
- Repeat the process until Y = V

Prim's Algorithm (basic idea)

F = ∅
Y = {v_1} // can be any vertex
while (instance not solved) {
    select v in V – Y nearest to Y (i.e. min weight);
    //selection procedure and feasibility check
    add v to Y;
    add the connecting edge to F;
    if (Y == V)
        the instance is solved;   //solution check
}

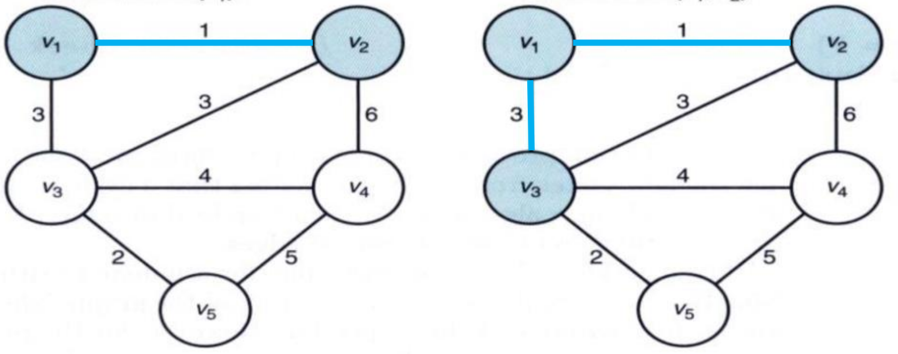
Example of Prim's Algorithm 1/3



Example of Prim's Algorithm 2/3

2. Vertex v_2 is selected because it is nearest to $\{v_1\}$.

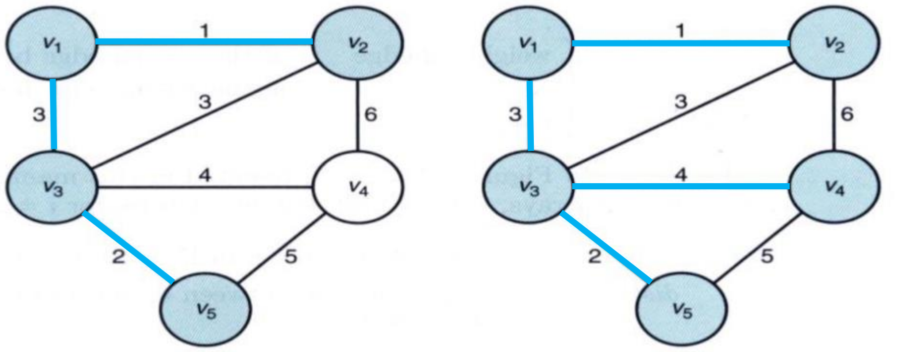
3. Vertex v_3 is selected because it is nearest to $\{v_1, v_2\}$.



Example of Prim's Algorithm 3/3

4. Vertex v_5 is selected because it is nearest to $\{v_1, v_2, v_3\}$.

5. Vertex v_4 is selected.



Prim's Algorithm (data structures)

- Adjacency Matrix Representation

$$W[i][j] = \begin{cases} weight & \text{If there is an edge from } v_i \text{ to } v_j \\ \infty & \text{If there is no edge from } v_i \text{ to } v_j \\ 0 & \text{If } i = j. \end{cases}$$

- Two additional arrays
  - *nearest*[*i*]: **index** of the vertex in *Y* nearest to v_i.
  - *distance*[*i*]: **weight** on edge between v_i and the vertex indexed by *nearest*[*i*].(i.e. weight of edge between v_i and $v_{nearest[i]}$)
- **Exercise**: Study Algorithm 4.1 (the Prim's algorithm).

Every-Case Time Complexity of Prim's Algorithm (Algorithm 4.1)

- **Input Size**: ***n*** (the number of vertices)
- **Basic Operation**: Two loops with n – 1 iterations inside the repeat loop.
- Repeat loop has n-1 iterations
- **Time complexity**:
  - $T(n) = 2(n-1)(n-1) \in \Theta(n^2)$

Is Prim's Algorithm Optimal?

- In **dynamic programming**, we only need to show that **principle of optimality** applies.
- Greedy algorithms are:
  - easier to develop (no need to establish recursive property and principle of optimality)
  - must formally prove optimal solution always produced
- We prove the optimality of the Prim's algorithm in two parts: Lemma 4.1 and Theorem 4.1

Optimality Proof of Prim's Algorithm

- In $G = (V, E)$, a subset F of E is called promising if edges can be added to it so as to form a MST. (i.e. F is part of an optimal solution.)
- **Lemma 4.1**: Let G be a connected, weighted, undirected graph; let F be a promising subset of E; and let Y be the set of vertices connected by the edges in F (i.e. (Y, F) is a subgraph of G). If e is an edge of minimum weight that connects a vertex in Y to a vertex in $V–Y$, then $F \cup \{e\}$ is promising. (i.e. expanding a promising set F with a minimum weight edge e is still promising)

Optimality Proof of Prim's Algorithm

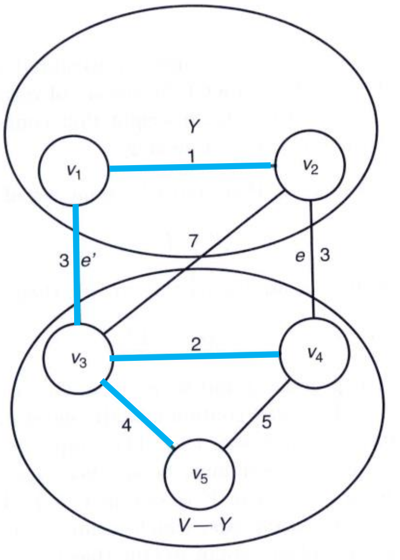
- **Theorem 4.1**: Prim's algorithm always produces a MST. (prove by induction)
- **Induction Base**: Clearly the empty set is promising.
- **Induction Hypothesis** : Assume that, after a given iteration of the loop, the set of edges so far selected, namely F, is promising.
- **Induction Step**: Let's show that the $F \cup \{e\}$ is promising. Because the edge e selected in the next iteration is an edge of minimum weight that connects Y to $V–Y$, $F \cup \{e\}$ is promising, by Lemma 4.1

Proof for Lemma 4.1

- Because F is promising, there must be some set of edges F' such that $F \subseteq F'$ and (V, F') is a MST.
- **Case 1**: If $e \in F'$, $F \cup \{e\} \subseteq F' \Rightarrow F \cup \{e\}$ is **promising**.
- **Case 2**: If $e \notin F'$,
  - $F' \cup \{e\}$ must form at least a cycle containing e. (why?)
  - There must be another edge $e' \in F'$ that connects Y to $V–Y$.
  - Cycle disappears if $F' \cup \{e\} – \{e'\} \Rightarrow$ a spanning tree.
  - Since e is minimum, the weight of $e \leq$ weight of e' (in fact, they must be equal). So, $F' \cup \{e\} – \{e'\}$ is an MST.
  - $F \cup \{e\} \subseteq F' \cup \{e\} – \{e'\}$, because e' cannot be in F (recall that edges in F connect only vertices in Y.)
  - Therefore, $F \cup \{e\}$ is **promising**, which completes the proof of Lemma 4.1.

Lemma 4.1 Illustrated

- Edges in ***F'*** are shaded in color.



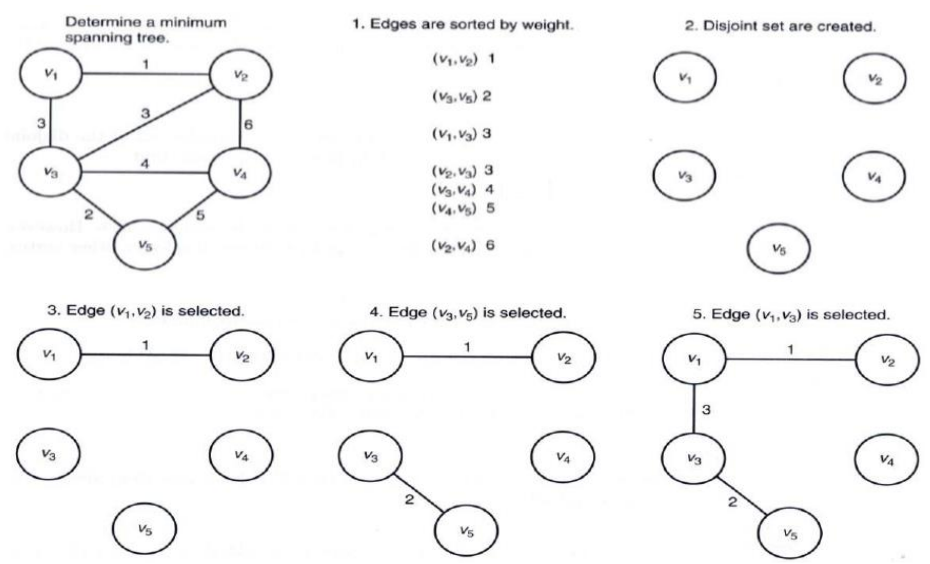
Kruskal's MSP Algorithm

- Create *n* disjoint subsets of *V* – one for every $v \in V$
- Each subset contains only one vertex
- Inspect edges according to non-decreasing weight. If an edge connects two vertices in disjoint subsets, add edge to final edge set and merge the two subsets
- Repeat until all subsets are merged into a single set

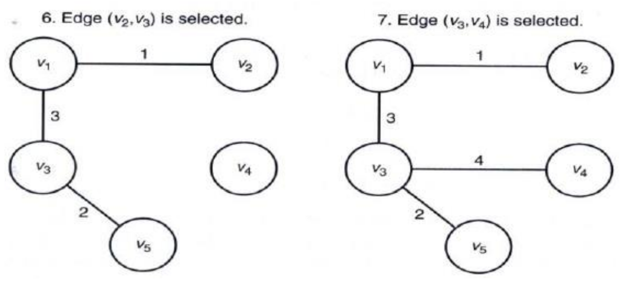
Kruskal's Algorithm

```
F:= 0;  // the set of edges
Create disjoint subsets of V, one for each vertex and
containing only that vertex;
Sort the edges in E in nondecreasing order;
while ( the instance is not solved ) {
    select next edge;   // selection procedure
    if (the edge connects two vertices in disjoint subsets) {
                                                  // feasibility check
        merge the subsets;
        add the edge to P;
    }
    if (all the subsets are merged)   // solution check
        the instance is solved;
}
```


Example of Kruskal's Algorithm



Example of Kruskal's Algorithm



- Note that in Step 6, (v_2, v_3) is selected first but rejected by the feasibility check. (why?)

Time Complexity Analysis of Kruskal's Algorithm

- **Basic operation**: a comparison operation.
- **Input size**: *n, m*, the number of vertices and edges, respectively.
- The time to sort the edges. (Recall the mergesort algorithm) $\Theta(m \lg m)$
- The time in the while loop: $\Theta(m \lg m)$
- The time to initialize n disjoint sets: $\Theta(n)$
- Since $m \geq n$-1, $W(m, n) = \Theta(m \lg m)$
- But in worst case (fully connected), $m = \frac{n(n-1)}{2} \in \Theta(n^2)$

$$W(m,n) \in \Theta(n^2 \lg n^2) = \Theta(2n^2 \lg n) = \Theta(n^2 \lg n)$$

- Optimality proof is very similar to Prim's case.

Is Kruskal's Algorithm Optimal?

Lemma 4.2:

- $G = (V, E)$ be a connected, weighted, undirected graph
- F is a promising subset of E
- Let e be an edge of minimum weight in $E - F$
- $F \cup \{e\}$ has no cycles
- $\Rightarrow F \cup \{e\}$ is promising
- Proof of Lemma 4.2 is similar to proof of Lemma 4.1.

Optimality Proof of Kruskal's Alg

- **Theorem 4.1**: Kruskal's Algorithm always produces a minimum spanning tree.
- **Proof**: use induction to show the set F is promising after each iteration of the repeat loop.
- **Induction base**: $F = \varnothing$ empty set is promising
- **Induction hypothesis**: assume after the ith iteration of the repeat loop, the set of edges F selected so far is promising
- **Induction step**: Show $F \cup \{e\}$ is promising where e is the selected edge in the i+1 th iteration

Optimality Proof of Kruskal's Alg

- *e* selected in next iteration, it has a minimum weight
- *e* connects vertices in disjoint sets
- Because *e* is selected, it is minimum and connects two vertices in disjoint sets
- By Lemma 4.2, $F \cup \{e\}$ is promising
- This completes the induction step which completes the optimality proof.

Comparison of Prim's and KrusKal's Algorithms

- Sparse graph:
  - *m* close to *n* – 1
  - Kruskal θ(n lg n) faster than Prim
- Highly connected graph
  - Kruskal θ(n^2 lg n)
  - Prim's faster

	$W(m,n)$	sparse graph	dense graph
Prim	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Kruskal	$\Theta(m \lg m)$ and $\Theta(n^2 \lg n)$	$\Theta(n \lg n)$	$\Theta(n^2 \lg n)$

Single-Source Shortest Paths Problem

- In Section 3.2, we have discussed a $\Theta(n^3)$ algorithm for finding the shortest paths between ALL pairs of vertices in a graph.
- The Single-Source Shortest Paths Problem (SSSP) is to find the shortest paths from ONE particular vertices to all other vertices.
- Dijkstra(1959) developed a $\Theta(n^2)$ greedy algorithm to solve this problem.
- We will assume that there is a path from the vertex to all others (i.e. connected).
- It's easy to modify the algorithm if this is not so.

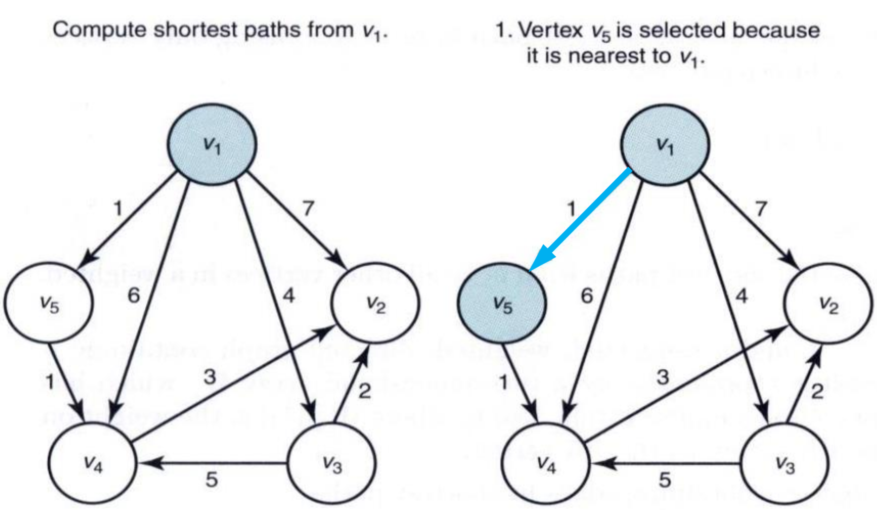
Dijkstra's SSSP Algorithm

- The algorithm is very similar to Prim's algorithm for Minimum Spanning Tree. We include edges to the solution set one-by-one with a greedy strategy.
- Only works for non-negative weight edges.
- Complexity analysis and proof are also similar to Prim's Algorithm.
- A $\Theta(n^2)$ algorithm.

Dijkstra's SSSP Algorithm

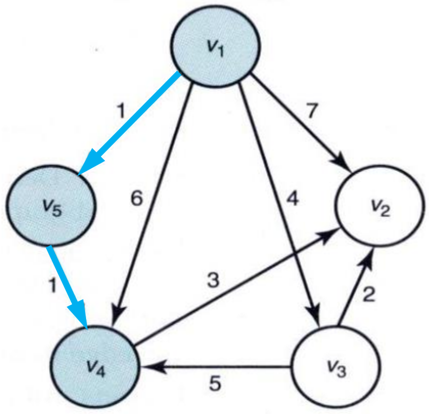
```
Y = {v1};  // assume that v1 is the starting vertex
F = ∅;  // the edges in shortest paths
while ( the instance is not solved ) {
    select a vertex v from V – Y that has the
    shortest path from v1 using only vertices
    in Y as intermediates;
    add v to Y;
    add the edge (on the shortest path) that
    touches v to F;
    if ( Y == V ) then the instance is solved;
}
```


Example of Dijkstra's Algorithm 1/3

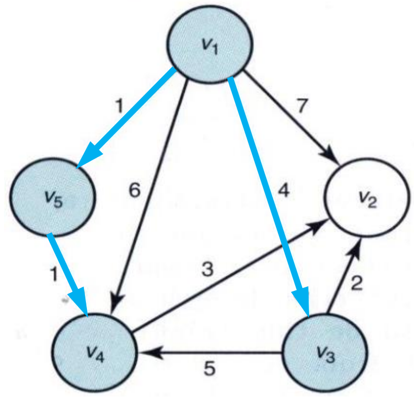


Example of Dijkstra's Algorithm 2/3

2. Vertex v_4 is selected because it has the shortest path from v_1 using only vertices in $\{v_5\}$ as intermediates.

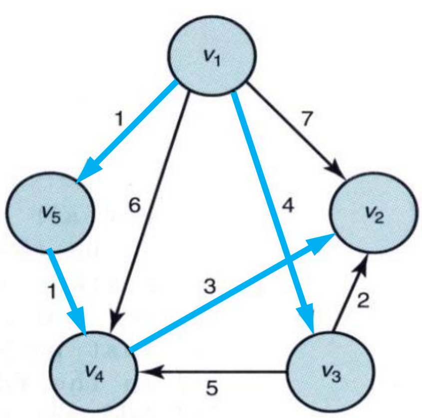


3. Vertex v_3 is selected because it has the shortest path from v_1 using only vertices in $\{v_4, v_5\}$ as intermediates.



Example of Dijkstra's Algorithm 3/3

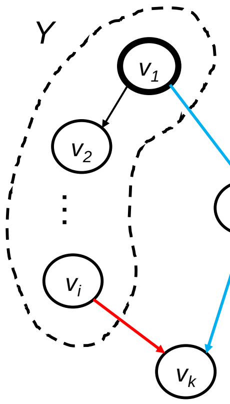
4. The shortest path from v_1 to v_2 is $[v_1, v_5, v_4, v_2]$.



- It is important to note that in each iteration, we always select the vertex that is nearest to v_1 through Y.
- After inclusion, a vertex or edge remains in Y or F till the end.

Why does Dijkstra's Algorithm work?

- Is it possible to have the following situation? i.e. when we select the shortest path to v_k through Y there exists a shorter path to v_k not through Y ?



Analysis/Proof of Dijkstra's Algorithm

- The algorithm has identical control structure as the Prim's MST algorithm.
- The complexity is: $T(n) = 2(n - 1)^2 \in \Theta(n^2)$.
- We can also prove that the algorithm always produces shortest paths. (i.e. optimal)
- The proof is similar to Prim's algorithm. (exercise)
- The algorithm can also be implemented using a heap or a Fibonacci heap to obtain a $\Theta(m \lg n)$ or a $\Theta(m + n \lg n)$ algorithm, respectively. (see Textbook for details)

Greedy vs Dynamic Programming

- Both are good for solving optimization problems.
- Shortest Path
  - Floyd – all pairs (dynamic programming)
  - Dijkstra – single source (greedy)
- Greedy algorithms are usually simpler.
- Greedy algorithms do not always produce optimal solution – must formally prove it.
- Dynamic Programming – must show that the principle of optimality applies.

Scheduling Problems

- A hair stylist has several customers waiting for different treatments.
- Different treatments take different amount of time, but the stylist knows how long each takes.
- The goal is to **schedule** the customers in such a way as to **minimize** the total time they spend both waiting and being served (called the time in the system).
- The scheduling problem to minimize the total time in the system has many applications. (Can you think of any examples?)

Scheduling with Deadlines

- Each job (customer) takes the same amount of time to complete.
- Each job has a **deadline** by which it **must start** to yield a profit associated with that job.
- A deadline means a DEADline ! Executing a job whose deadline has passed is meaningless(w/o profit). (May even have a penalty !)
- The goal is to schedule the jobs to maximize the total profit. (It may not be possible to meet all deadlines.)
- The Scheduling with Deadlines problem also has many applications that should be easy to come by.
- We will leave this part for your to study.

Example of Scheduling Problem

- Three jobs and the service times:

  $t_1 = 5, \quad t_2 = 10, \quad t_3 = 4$

- If we schedule them in order 1, 2, 3, then

Job	Time in the System
1	5 (service time)
2	5 (wait for job 1) + 10 (service time)
3	5 (wait for job 1) + 10 (wait for job 2) + 4 (service time)

- The total time is

$$\underbrace{5}_{\text{Time for job 1}} + \underbrace{(5+10)}_{\text{Time for job 2}} + \underbrace{(5+10+4)}_{\text{Time for job 3}} = 39$$

Example of Scheduling Problem

- List all possible schedules and total times:

Schedule	Total Time in the System
[1, 2, 3]	5 + (5 + 10) + (5 + 10 + 4) = 39
[1, 3, 2]	5 + (5 + 4) + (5 + 4 + 10) = 33
[2, 1, 3]	10 + (10 + 5) + (10 + 5 + 4) = 44
[2, 3, 1]	10 + (10 + 4) + (10 + 4 + 5) = 43
[3, 1, 2]	4 + (4 + 5) + (4 + 5 + 10) = 32
[3, 2, 1]	4 + (4 + 10) + (4 + 10 + 5) = 37

- Schedule [3, 1, 2] is optimal with total time 32.
- "Shorter job first" seems to be good strategy.

Greedy Scheduling Algorithm

1. Sort the jobs by service time in nondecreasing order;
2. while (the instance is not solved) {
3. schedule the next job;
4. if (there are no more jobs)
5. the instance is solved;
6. }

Analysis of Greedy Scheduling Algorithm

- It is easy to see that the main part is to sort the jobs according to service time. Therefore

  $W(n) \in \Theta(n \lg n)$
- It can be shown that schedule produced by the algorithm is optimal. (exercise)
- While the optimality of the algorithm need to be proved as usual, we can get a stronger result that this schedule is the only optimal one.

Optimality Proof of Greedy Scheduling

- **Theorem 4.3**: The **only** schedule that minimizes the total time in the system is one that schedules jobs in nondecreasing order by service time.
- **Proof**: By contradiction.
  - Let t_i be the service time for the ith job scheduled in some particular optimal schedule with total time T.
  - If they are not scheduled in nondecreasing order, then for at least one i where $1 \le i \le n - 1$, $t_i > t_{i+1}$
  - Rearrange the original schedule by swapping the ith and $(i+1)$st jobs. Then the new total time $\boldsymbol{T'}$:

    $$T' = T + t_{i+1} - t_i < T$$

    which contradicts the optimality of T. (why?)

Multiple-Server Scheduling Problem

- We generalize the algorithm to handle Multiple-Server Scheduling problem with m servers.
- Order the jobs again by service time in nondecreasing order.
- Let the 1st server serve the 1st job, the 2nd server the 2nd job, … , and the mth server the mth job.
- The 1st server will finish first. (why?)
- Then, the 1st server serves the $(m+1)$st job. Similarly, the 2nd server serves the $(m+2)$nd job, and so on.

Multiple-Server Scheduling Problem

- The scheme is as follows:
  - Server 1 serves jobs 1, $1 + m$, $1 + 2m$, $1 + 3m$, …
  - Server 2 serves jobs 2, $2 + m$, $2 + 2m$, $3 + 3m$, …
  - …
  - Server i serves jobs i, $i + m$, $i + 2m$, $i + 3m$, …
  - …
  - Server m serves jobs m, $2m$, $3m$, $4m$, …
- The jobs are served in the order

  $1, 2, \ldots, m, 1 + m, 2 + m, \ldots, 2m, 1 + 2m, \ldots$

  which is optimal. (why?)

Activity Selection Problem (ASP)

- A classic problem for the greedy approach.
- **Problem**: Given a set of activities S={a_1, a_2,…, a_n}
  - Execution of activities cannot overlap. (They use resources, such as lecture hall, one at a time.)
  - Each a_i, has a start time s_i, and finish time f_i, with $0\leq s_i< f_i<\infty$.
  - a_i and a_j are compatible if $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- **Goal**: select maximum-size subset of mutually compatible activities.

Example and Characteristics

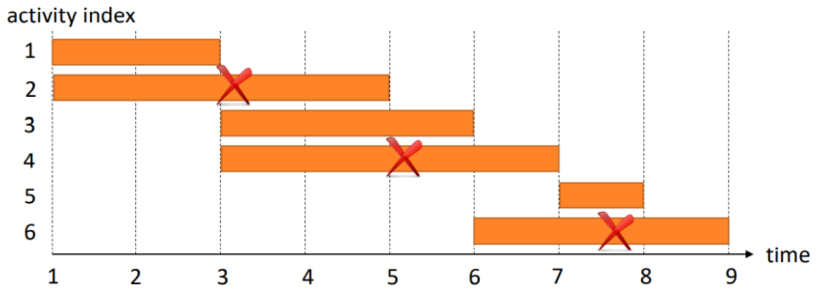
- An **example** of the activity selection problem:

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

- **Characteristics** of the activity selection problem:
  - Overlapping activities cannot be selected together. (e.g a_1 and a_2)
  - If the time interval of an activity is completely contained in another activity, it is always better to select the shorter one. (a_1 and a_2 are both in a_3. Choosing a_3 prohibits the selection of a_1, a_2, a_4, a_5, a_6. Choosing a_1 or a_2 only prohibits a_5.)

Interval Scheduling

- Activity selection problem is also known as interval scheduling problem since we are to schedule max number of non-overlapping intervals.



DP vs Greedy Strategy for ASP

- We will consider ASP with both DP and greedy approaches.
- We analyze the optimal substructure property of ASP.
- Then a DP solution is attempted.
- We will see that a greedy approach can be derived from the DP approach.

Optimal Substructure of ASP

- Define $S_{ij}=\{a_k \in S : f_i \leq s_k < f_k \leq s_j\}$,
  S_{ij} is the subset of activities in S that can start after activity a_i finishes and finish before activity a_j starts.
- Define f_0=0 and $s_{n+1} = \infty$. Then $S = S_{0,n+1}$, and the ranges for i and j are given by $0 \leq i, j \leq n+1$.
- An optimal solution including a_k to S_{ij} contains within it the optimal solution to S_{ik} and S_{kj}.
- The activities selection: $A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj}$

Recursive Property of ASP

- The activity a_k can be any one in S_{ij}.
- Assume c[i, j] is the number of activities in a maximum-size subset of mutually compatible activities in S_{ij}. So the solution is c[0, n+1] of $S_{0,n+1}$ (= S).

$$c[i, j] = c[i,k] + c[k, j] + 1$$

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

Converting DP Sol to Greedy Sol

- The previous formulation can be solve with DP.
- A greedy strategy can be derived from it.
- **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
  1. Activity a_m is used in some maximum-size subset of mutually compatible activities of S_{ij}.
  2. The subproblem S_{im} is empty, so that choosing a_m leaves the subproblem S_{mj} as the only one that may be nonempty.

Greedy Strategy to ASP

- To solve S_{ij}, choose a_m in S_{ij} with the earliest finish time, then solve S_{mj}, (S_{im} is empty, based on the Theorem)
- It is certain that optimal solution to S_{mj} is in optimal solution to S_{ij}.
- No need to solve S_{mj} ahead of S_{ij}.
- Therefore a top-down solution can be derived.

Recursive Greedy Algorithm

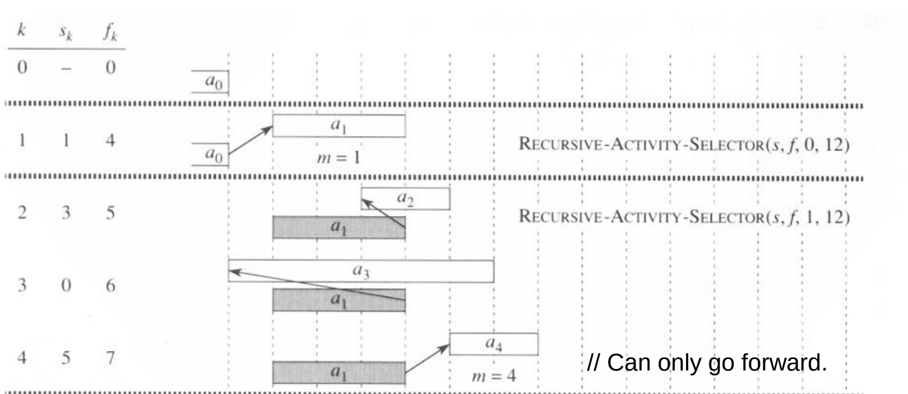
RECURSIVE-ACTIVITY-SELECTOR(s, f, i, j)

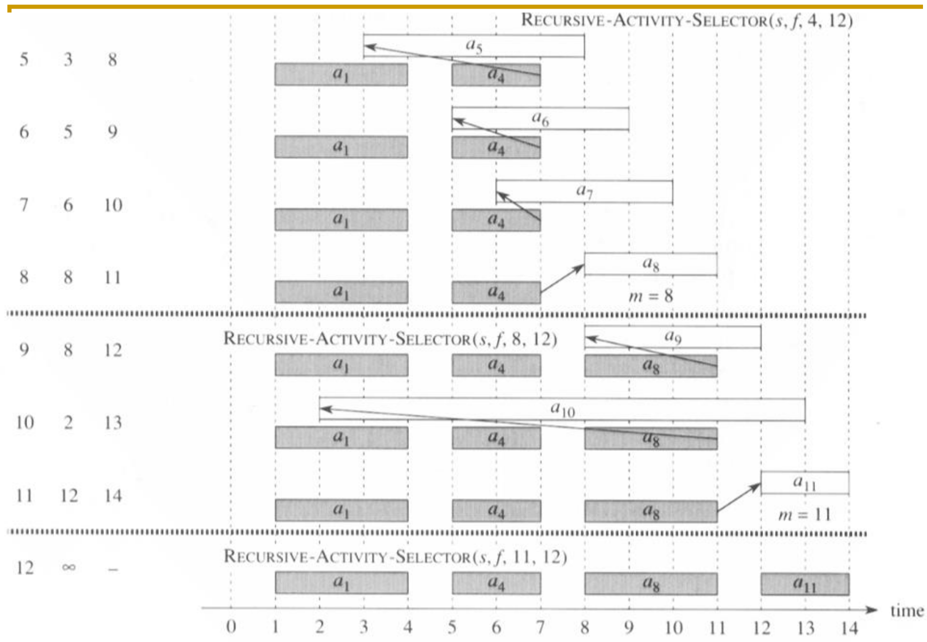
```
m ← i + 1
while m < j and s_m < f_i        // Find the first activity in S_ij
      do m ← m + 1
if m < j
      then return {a_m} ∪ RAS(s, f, m, j)
      else return 0
```


Example of RAS Execution

- Given

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





Iterative Greedy Algorithm

GREEDY-ACTIVITY-SELECTOR(s, f)

$n \leftarrow length[s]$
$a \leftarrow \{a_1\}$
$i \leftarrow 1$
for $m \leftarrow 2$ to n
  do if $s_m \geq f_i$
    then $A \leftarrow A \cup \{a_m\}$
      $i \leftarrow m$
return A

The Knapsack Problem

- Problem description:
  - Given n items and a "knapsack."(背包)
  - Item i has weight $w_i > 0$ and has value $v_i > 0$.
  - Knapsack has capacity of W.
  - **Goal**: Fill knapsack so as to maximize total value w/o exceeding the capacity.
- Mathematical description:
  - Given two n-tuples of positive numbers $< v_1, v_2, \ldots, v_n >$ and $< w_1, w_2, \ldots, w_n >$, and $W > 0$, we wish to determine the subset $T \subseteq \{1,2,\ldots, n\}$ that

$$\text{maximize} \sum_{i \in T} v_i \qquad \text{subject to} \sum_{i \in T} w_i \le W$$

Example of Knapsack Problem

- Weight capacity W = 5kg.

i	v_i	w_i
1	\$10	1kg
2	\$12	1kg
3	\$15	2kg
4	\$20	3kg

- The possible ways to fill the knapsack:
  - {1, 2, 3} has value \$37 with weight 4kg.
  - {4, 3} has value \$35 with weight 5kg. (greedy)
  - {1, 2, 4} has value \$42 with weight 5kg. (optimal)
- The greedy approach by always selecting the item with highest value is not optimal.

Fractional Knapsack Problem

- The previous problem is also called the 0-1 knapsack problem.
  - Each item can only be taken or not taken as a whole.
- Now, we change the problem to enable one to take any fraction of the item.
  - Both weight and value follow the fraction.
  - This is called the fractional knapsack problem.
  - A greedy approach can be developed by always choosing the item with the largest **value-weight ratio**.
  - It can be shown that the greedy algorithm always produce optimal solution.

Example of Fractional Knapsack Problem

- Weight capacity W = 5kg.

i	v_i	w_i	v_i/w_i
1	$10	1kg	10$/kg
2	$12	1kg	12$/kg
3	$15	2kg	7.5$/kg
4	$20	3kg	6.67$/kg

- By the greedy approach:
  - Take item 2: remain 4kg and total value is 12.
  - Take item 1: remain 3kg and total value is 22.
  - Take item 3: remain 1kg and total value is 37.
  - Take 1/3 of item 4: remain 0kg and total value is 43.67.
- It is optimal. Try to prove it.

Greedy Approach vs Dynamic Prog

- **Common**: find optimal solution for subinstances of the problem.
- **Difference**:
  - **Greedy**: any optimal solution for subinstance is a part of the final optimal solution.
  - **Dynamic Programming**: only a subset of optimal solution for subinstances construct the final optimal solution.
- Different approaches are used for similar problems with only little difference.
  - Shortest path problem vs single-source shortest path problem.
  - 0-1 knapsack problem vs fractional knapsack problem.
- Analyzing the problem and choosing the best strategy is important in algorithm design.

When to use/avoid Greedy Approach

- Keep in mind that greedy algorithms do not always produce optimal solutions.
- Problems that can be solved with greedy approach and produce optimal solutions:
  1. Optimal substructure: an optimal solution can be constructed from optimal solutions of its subproblems.
  2. Optimality of greedy-choice: each greedy step maintains the optimality of the solution set.
- Otherwise, greedy approach may be suboptimal.
- In such cases, if the problem exhibits overlapping subproblems, then use **dynamic programming**.

Assignment 4: Greedy Approach

1. Assume that there are m cars and n parking lots initially aligned along a straight road on specific locations. A car can stay at its location, move from x to x+1, or move from x to x-1. Each move takes 1 minute. Assign cars to parking lots so that the time the last car gets to its lot is minimized.
   - Design a greedy algorithm to solve the problem.
   - Prove the optimality of your algorithm.
   - Analyze the time complexity of your algorithm.
   - Write a program and test it properly.

Assignment 4: Greedy Approach

2. Given a set of n keywords $S = \{k_1, k_2, \ldots, k_n\}$ where no word is a part of another word. Find a sentence string T such that for all $k_i \in S \Rightarrow k_i \in T$ and $|T|$ (i.e. the length of T) is minimized.
   - Design a greedy algorithm to solve the problem.
   - Prove the optimality of your algorithm.
   - Analyze the time complexity of your algorithm.
   - Write a program and test it properly.
3. Textbook exercises: Chapter 4, exercises 2, 7, 13, 19, 22

Due date: three weeks.