# Lecture Note 3

## CSCI 6470 Algorithms
## (Fall 2024)

Liming Cai

School of Computing UGA

September 30, 2024

# Chapter 4. Advanced Algorithmic Techniques

Topics to be discussed:

- Dynamic programming
- Greedy algorithms
- Flow networks

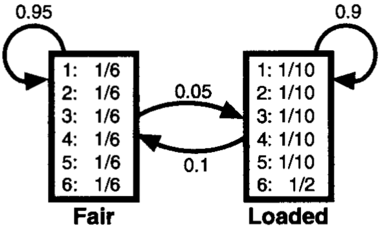
# 1. Dynamic programming

Introduction to DP with problem:
computing the $n^{\mathrm{th}}$ Fibonacci numbers

- naive recursive algorithm (top-down), at least $1.41^n$
  (how to describe "at least $1.41^n$"?)
- memoized recursive algorithm (top-down, use lookup table) $O(n)$
- iterative algorithm (bottom-up) $O(n)$

# 1. Dynamic programming

Decoding dishonest dice rollings




*A hidden Markov model M*

```
O = 1654622316516643254132565442355122126161626  <- observable
S = FFFFFFFFFLLLLLLLFFFFFFFFFFFFFFFFFFFLLLLLLLL  <- hidden dice
```

- decoding question: what are the underlying sequence of dices used?

# 1. Dynamic programming

A more significant problem:

# 1. Dynamic programming

A more significant problem:

# 1. Dynamic programming

Intuitively,

- dynamic programming is an exhaustive search method;
- dynamic programming fills a table(s) with numerical data according to certain order;
- data dependency order in the table implies the desired solution;

# 1. Dynamic programming

Problem 1: Single-source shortest paths in DAG

- (based on topological sort order), recall how we did it;
- a slightly different order,

**for** $v = 1$ **to** $n$ (order in a topological sort)
$$dist(v) = \min_{(u,v)\in E} \{dist(u) + l(u,v)\}$$
remember the corresponding `prev`

- how to write this into pseudo code?

# 1. Dynamic programming

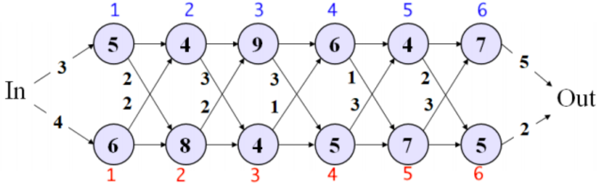
- Fill the table `dist` in a topological order

```
for v = 1 to n
   dist(v) = infinite;
   prev(v) = nil;
   for all (u, v) in E
      if dist(v) > dist(u) + l(u,v)
         dist(v) = dist(u) + l(u,v);
         prev(v) = u;
```

- Print out all shortest-paths based on dist and prev
  [in class exercise]

# 1. Dynamic programming

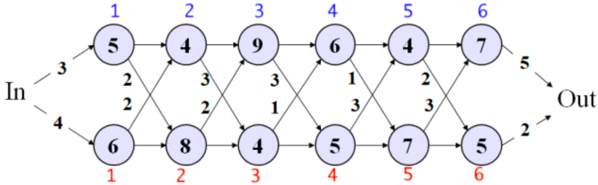
<u>Problem 2</u>: the fastest path through a factory




- $2n$ stations; each station has processing time;
- no time cost for transitions within the same production line;
- there are time costs between two different production lines;
- a path time $=$ sum of all processing and transition times on the path;

# 1. Dynamic programming

**Step 1: analysis of the problem**




- the fastest path `In` $\rightsquigarrow$ `Out` has to be

  the faster of $\begin{cases} \text{a fastest path } \texttt{In} \rightsquigarrow 6 \text{ then edge } 6 \to \texttt{Out}, \\ \text{a fastest path } \texttt{In} \rightsquigarrow 6 \text{ then edge } 6 \to \texttt{Out} \end{cases}$

- the fastest path `In` $\rightsquigarrow$ 4 has to be

  the faster of $\begin{cases} \text{a fastest path } \texttt{In} \rightsquigarrow 3 \text{ then edge } 3 \to 4, \\ \text{a fastest path } \texttt{In} \rightsquigarrow 3 \text{ then edge } 3 \to 4 \end{cases}$

# 1. Dynamic programming

In general,

- for every $k = 2, 3, \ldots, n$,
  the fastest path $\texttt{In} \rightsquigarrow k$ has to be
  $$\text{the faster of} \begin{cases} \text{a fastest path } \texttt{In} \rightsquigarrow k-1 \text{ then } k-1 \rightarrow k, \\ \text{a fastest path } \texttt{In} \rightsquigarrow k-1 \text{ then } k-1 \rightarrow k \end{cases}$$

- for every $k = 2, 3, \ldots, n$,
  the fastest path $\texttt{In} \rightsquigarrow k$ has to be
  $$\text{the faster of} \begin{cases} \text{a fastest path } \texttt{In} \rightsquigarrow k-1 \text{ then edge } k-1 \rightarrow k, \\ \text{a fastest path } \texttt{In} \rightsquigarrow k-1 \text{ then } k-1 \rightarrow k \end{cases}$$

- what about $k = 1$?

  the fastest path $\texttt{In} \rightsquigarrow 1$ is $\texttt{In} \rightarrow 1$

  the fastest path $\texttt{In} \rightsquigarrow 1$ is $\texttt{In} \rightarrow 1$

# 1. Dynamic programming

Now what?

Two observations:

- the problem is to find a shortest path from station `In`;
  every path is associated with a time (`dist`);
- shortest paths are recursively defined;
  so fastest times can be recursively defined;

# 1. Dynamic programming

**Step 2: define numerical objective function**

For $k = 1, 2, \ldots, n$, $i = 1, 2$:

- Label with $(1,1), \ldots, (1,n)$ for stations in production line 1; and with $(2,1), \ldots, (2,n)$ for production line 2;
- Let $pt_i(k)$ be the processing time on station $(i,k)$;
- Let $tt_i(k-1)$ be the transfer time from station $(i, k-1)$ to station $(\widetilde{i}, k)$, where $\widetilde{i}$ is the opposite production line of $i$;
- Define **function** $ft_i(k)$ to be the fastest time of a path from station `In` to station $(i,k)$;

  Then

$$ft_i(k) = \min \begin{cases} ft_i(k-1) + pt_i(k) \\ ft_{\widetilde{i}}(k-1) + tt_{\widetilde{i}}(k-1) + pt_i(k) \end{cases} \qquad k \geq 2$$

$$ft_i(1) = \text{the known time from } \mathtt{I} \text{ to station } (i,1) + pt_i(1)$$

# 1. Dynamic programming

**Step 3: Establish and fill DP tables**

- establish a table $F_{2\times n}$ to store values of function $ft_i(k)$, where $i = 1, 2$ and $k = 1, 2, \ldots, n$;
- establish a table $\texttt{prev}_{2\times n}$ to store previous stations
- fill the tables using the recursive formulas for $ft_i(k)$, with an iterative program;
- write the pseudo code for table filling (in-class exercise)

# 1. Dynamic programming

**Step 4: Trace back the fastest path**

- table `prev` should contain enough information about the fastest path
- but wait, what is the fastest time through the factory?
- from the fastest time, we know the last station of which production line is on the fast path before station `Out`;
- traceback can start from that station, and recursively;
- write pseudo code for traceback (in-class exercise)

# 1. Dynamic programming

Complexity of a DP algorithm

- essentially the time to fill tables
  $=$ table size $\times$ cell filling time
- plus the time to trace back solution(s) (how much is it?)

# 1. Dynamic programming

**Characteristics** of problems that can be solved with DP:

(1) **Optimal substructures**

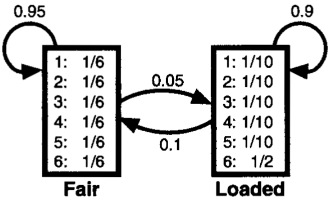
- the solution to the problem can be recursively constructed from solutions to some subproblems;
- solutions to subproblems should also be optimal;

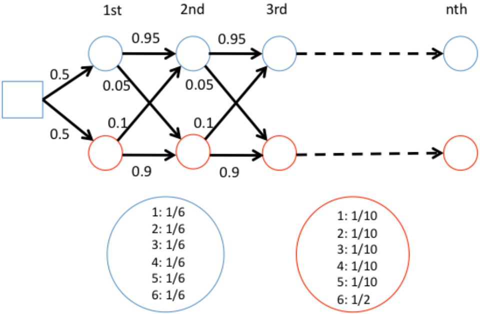
(2) **Overlapping subproblems**

- one subproblem solution is shared by more than one other problem to construct their solutions

# 1. Dynamic programming

Problem 3: Decoding dishonest dice rolls




$O = o_1 o_2 \dots o_n$ observed dice roll outcomes;

$S = d_1 d_2 \dots d_n$ the sequence of dice **with highest probability**

# 1. Dynamic programming

Probability of dice rollings:

- emission probability $e_F(k) = \frac{1}{6}$ for all $k = 1, 2, \ldots, 6$;
- transition probability

$$t_{FF} = 0.95, t_{FL} = 0.05, t_{LL} = 0.9, t_{LF} = 0.1$$

- computing probability of rolling 2466 with dice $\mathtt{FFLL}$

$$0.5 \times e_F(2) \times t_{FF} \times e_F(4) \times t_{FL} \times e_L(6) \times t_{LL} \times e_L(6) =?$$

is it different from with dice $\mathtt{FFFF}$ ? (in-class exercise)

# 1. Dynamic programming

**Step 1: problem analysis**

Analog between decoding dice and finding the fastest path through factory

- a sequence of dice consists of either F or L dices in each position;

  a path through factory consists of stations either in production line 1 or line 2;

- the most like sequence is one with the highest probability;

  the fastest path is one with smallest time;

# 1. Dynamic programming

- the most likely sequence ends at either `Fair` or `Loaded` die;
- for $k \geq 1$,
  the most likely sequence of length $k$ ending at `Fair` die is
  (1) either the most likely sequence of length $k-1$ ending at `Fair` die followed by `Fair` die,
  (2) or the most likely sequence of length $k-1$ end at `Loaded` die followed by `Fair`,

  whichever has higher probability

# 1. Dynamic programming

**Step 2: definition of objective function**

Define $m(k, F)$ to be the highest probability of a sequence of $k$ dice ending at `Fair` die to emit the first $k$ observed numbers. Then

Recursively,

$$m(k, F) = \max \begin{cases} m(k-1, F) \times t_{FF} \times e_F(o_k); \\ m(k-1, L) \times t_{LF} \times e_F(o_k); \end{cases}$$

$m(k, L) =?$ (in-class exercise)

base cases:

$m(1, F) = 0.5 \times e_F(o_1)$
$m(1, L) = 0.5 \times e_L(o_1)$

# 1. Dynamic programming

**Step 3: fill DP tables**

- what tables are needed?
- how to fill the tables?
- pseudo code for the table filling process (in-class exercise)

# 1. Dynamic programming

**Step 4: trace back solutions**

- what solutions?
- how to get the solutions?
- pseudo code for traceback (in-class exrecise)

# 1. Dynamic programming

The **Decoding dishonest dice** problem has the characteristics

- optimal substructure, what is it in the problem?
- overlapping subproblems, what are they in the problem?

# 1. Dynamic programming

Problem 4: Knapsack problem

# 1. Dynamic programming

Problem 4: Knapsack problem

- input: $n$ items, of size/weight $s_i$ and value $v_i$, $i = 1, \ldots, n$, and a knapsack of volume $W$;

  output: a subset of items $A \subseteq \{1, 2, \ldots, n\}$, such that

$$\sum_{i \in A} v_i \text{ is maximized, subject to } \sum_{i \in A} s_i \leq W$$

- there is a recursive solution to this problem.

# 1. Dynamic programming

**Step 1: problem analysis**

- in the previous three problems, subproblems are “prefixes”;
  do we have ”prefix subproblems” for Knapsack?
- how to select some items from the first $k$ items into a space of ? volume $X$, $X \leq W$.
- either item $k$ is selected, with gain of value $v_k$ but decrease of available space to $X - s_k$;
- or discard item $k$, with no change in value and no change in available space

# 1. Dynamic programming

**Step 2: define objective function**

- associated with a solution is the total value of selected items;
- define objective function $V(k, X)$ to be the maximum value of items selected from $\{1, 2, \dots k\}$. Then

$$V(k, X) = \max \begin{cases} V(k-1, X - s_k) + v_k & X \geq s_k \\ V(k-1, X) \end{cases}$$

base cases

$$V(0, X) = 0; \; X = 0, 1, 2 \dots, W$$

$$V(k, X) = 0; \; k = 0, 1, 2, \dots, n$$

# 1. Dynamic programming

## Step 3: Fill DP tables

- dimensions of tables: $(n+1) \times (W+1)$
- data dependence, `prev` info;
- fill the table, pseudo code (taken-home exercise)

| | X = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| k=0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| k=1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| k=2 | 0 | 0 | 1 | 2 | 2 | | | | |
| k=3 | 0 | | | | | | | | |
| k=4 | 0 | | | | | | | | |

| item | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| value | 1 | 2 | 5 | 6 |
| size | 2 | 3 | 4 | 5 |

W=8

**base cases**

$V(0, X) = 0,$
$V(k, 0) = 0$

**recursive cases**

$$V(1,1) = \max \begin{cases} V(0, 1-s1) + v1 & \text{not possible} \\ V(0, 1) = 0 \end{cases}$$

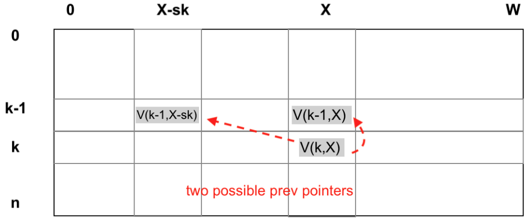
$$V(2,3) = \max \begin{cases} V(1, 3-s2) + v2 = V(1, 0) + 2 = 2 \\ V(1, 3) = 1 \end{cases}$$

# 1. Dynamic programming

**Step 4. Trace back optimal packing**

- How to identify `prev` info? recall

$$V(k,X) = \max \begin{cases} V(k-1, X-s_k) + v_k & X \geq s_k \\ V(k-1, X) \end{cases}$$




- pseudo code for traceback of optimal solution from DP tables

# 1. Dynamic programming

Problem 5: Edit Distance problem

measuring distance between two input strings, based on how many

(1) matches;
(2) insertions;
(3) deletions;
(4) mismatches;

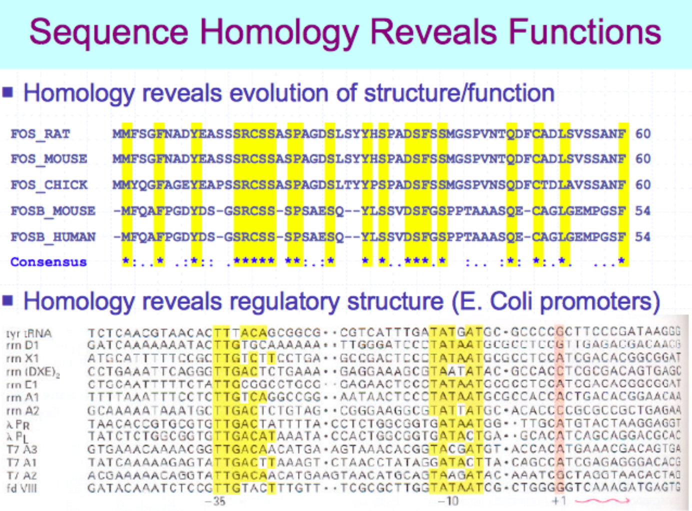
```
E V O L V I N G          edited    _ E V O L V _ I _ N G
R E V O L U T I O N       ==>      R E V O L U T I O N _
```

- scores is a part of input;
  e.g., match 0, insertion/deletion 1, mismatch 2.
  the above edit gives 6 points. Is there a lower score edit?
- **the goal of the problem is to find a lowest score edit.**

# 1. Dynamic programming

Problem 5: Edit Distance problem

A significant application: biological sequence alignment



# 1. Dynamic programming

**Step 1 identify optimal substructure**

Handle the problem recursively:

```
E V O L V I N [G]          3 possible    G    _    G
R E V O L U T I O [N]      scenarios     N    N    _
```

3 subproblems: to find lowest score edits for

```
(1)  E V O L V I N          G
     R E V O L U T I O      N

(2)  E V O L V I N G        _
     R E V O L U T I O      N

(3)  E V O L V I N          G
     R E V O L U T I O N    _
```

Lowest score edit is chosen over the 3 subproblems.

# 1. Dynamic programming

**Step 2 define objective function**

- input two strings $x[1..m]$ and $y[1..n]$;
- define $E(i,j)$ be the smallest distance (lowest score) between prefixes $x[1..i]$ and $y[1..j]$;
- then the recursive formula for $E(i,j)$:

$$E(i,j) = \min \begin{cases} E(i-1, j-1) + \texttt{diff}(i,j) \\ E(i, j-1) + 1 \\ E(i-1, j) + 1 \end{cases}$$

where

$$\texttt{diff}(i,j) = \begin{cases} 0 & x[i] = y[j] \\ 2 & x[i] \neq y[j] \end{cases}$$

`diff` and all scores can be redefined for other problems!

# 1. Dynamic programming

**Step 3 DP table filling** (taken-home exercise)

**Step 4 Solution trace back** (taken-home exercise)

# 1. Dynamic programming

Pseudocode for DP table filling and solution trace back

- DP table filling:

  (1) design of tables,
  (2) order of cells to fill,
  (3) iterative (can it be recursive?)

- Trace back:

  repetitive process following `prev` information (can it be recursive?)

# 1. Dynamic programming

Example pseudocode for DP, for problem **Two-Assembly Lines**

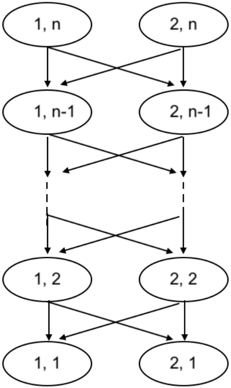
```
 function fastest-time(pt, tt, n)
   ft[1,1] = tt_In(1); ft[1,2] = tt_In(2);
   for k = 2 to n do
     ft[1,k] = pt_1(k) + min{ft[1,k-1],ft[2,k-1] + tt_2(k-1)};
     ft[2,k] = pt_2(k) + min{ft[2,k-1],ft[1,k-1] + tt_1(k-1)};
     prev[1,k] and prev[2,k] store corresponding info, either 1 or 2;
   ft[Out] = min{ft[1,n] + tt_1(n), ft[2,n] + tt_2(n)};
   prev[Out] stores corresponding info, 1 or 2;
   return (ft, prev);

function trace-back-main(prev, n);
   i = prev[Out]; k = n;
   call trace(i, k);
   print(i, "--> Out");

function trace(prev, i, k);
   if k=1 then print("In -->", i);
   else
       trace(prev, prev[i,k], k-1);
       print(prev[i,k], "-->", i);
```

# 1. Dynamic programming

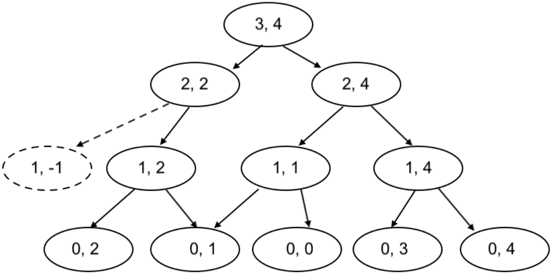
Exploitation of problem structure

- problem decomposition (always top-down, recursive structure)
- top-down implementation (mostly recursive)
- bottom-up implementation (mostly iterative)
- where does DP fit?
- memoization (using more space to gain time speed up)
- memoized DP

# 1. Dynamic programming




Assembly Line

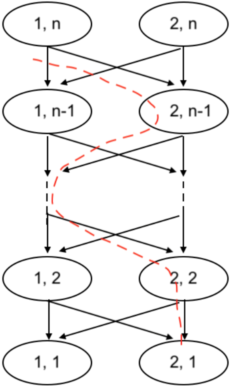



| item | size | value |
|---|---|---|
| 1 | 1 | 20 |
| 2 | 3 | 40 |
| 3 | 2 | 50 |

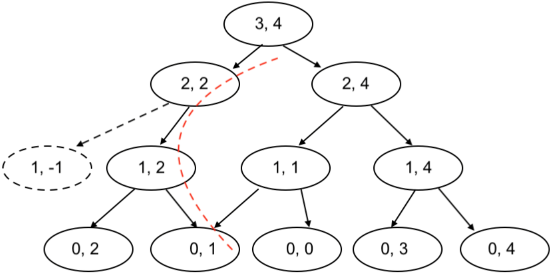
W = 4

0-1 Knapsack

# 1. Dynamic programming




Assembly Line




| item | size | value |
|---|---|---|
| 1 | 1 | 20 |
| 2 | 3 | 40 |
| 3 | 2 | 50 |

W = 4

0-1 Knapsack

# 2. Greedy algorithms

**Problem 1**: Fractional Knapsack problem

- input: $n$ items, of value $v_i$ and size $s_i$, and knapsack size $W$;
- output: $f_1, f_2, \ldots, f_n$, $0 \leq f_i \leq 1$, such that

$$\sum_{i=1}^{n} f_i v_i \text{ is maximized}$$

subject to $\sum_{i=1}^{n} f_i s_i \leq W$

Not only options of items and but also options of fractions!

# 2. Greedy algorithms

There are **greedy algorithms** for this problem.

(1) Idea:

- compute "value density" $d_i = \frac{v_i}{s_i}$;
- sort items according to $d_i$, non-decreasingly;
- choose items in this order;
  pack the current item (whole or a part, space allowed)

(2) Efficient: only linear time is required + sorting time;

(3) unlike DP which guarantees an optimal solution,
  a greedy algorithm may not;

- we need to prove the greedy strategy leads to optimal solutions.

## 2. Greedy algorithms

A **greedy-choice property** for Fractional Knapsack:

The item with the maximum value density is in some optimal solution.

Proof:

- Assume items are ordered 1 to n based on density $d_i$;
- Let $S = \{f_1, f_2, \dots, f_n\}$ be any solution.
- If $f_1 < 1$, going through all $f_i, i = 2, \dots, n$, to update

$$f_1 = f_1 + \min\{1 - f_1, \frac{f_i s_i}{s_1}\}, \text{ and reduce } f_i \text{ accordingly}$$

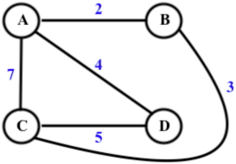
  until $f_1 = 1$ or $f_1 \times s_1 = W$.

- This yields solution $S'$, which contains item 1 and is optimal

# 2. Greedy algorithms

**Problem 2**: Minimum spanning tree (MST)

- what is a spanning tree of a graph $G$?




- significance of spanning tree and MST

# 2. Greedy algorithms

- Again **greedy-choice property**

  A problem has a **greedy choice property** if its optimal solution is computed from only one specific choice.

- MST problem has a greedy choice property.

# 2. Greedy algorithms

Intuitively,

- an edge with the smallest weight should be in some m.s.t.
  why?
- is this alway true as the idea being used repetitively?
  when does it not work?
- more precise terms are needed.

# 2. Greedy algorithms

Some terminologies:

- a **cut** in a graph $G = (V, E)$ is a partition of set $V$ into two:

$$(S, V - S), \text{ where } S \subset V, (S \neq \emptyset)$$

- an edge $(u, v)$ **crosses** cut $(S, V - S)$ if $u \in S, v \in V - S$;
- an edge is a **light edge** crossing a cut it is of the smallest weight among all edges that cross the cut.

# 2. Greedy algorithms

**Theorem**: A greedy-choice property for MST problem:

> *Let $G$ be a given graph. Then any light edge crossing any cut of the graph is in some minimum spanning tree of the graph.*

**Proof**: (using the Exchange method)

- let $T$ be an m.s.t. for $G$ and $(u, v)$ is a light edge crossing some cut $(S, V - S)$;
- if $(u, v)$ is in $T$, then the theorem is proved;
- otherwise, let edge $(x, y)$ in $T$ that crosses the cut $(S, V - S)$;
- then $T \cup \{(u, v)\}$ contains a cycle; why?
- let $T' = T \cup \{(u, v)\} - \{(x, y)\}$. Then $T'$ is a spanning tree; why
- because $(x, y)$ and $(u, v)$ cross $(S, V - S)$ and $(u, v)$ is a light edge, $T'$ is also m.s.t. for $G$. why?
- Because $T'$ contains $(u, v)$, the theorem is proved.

# 2. Greedy algorithms

Based on the greedy-choice property, if we can identify a light edge crossing some cut (any cut), then we can safely add the edge into partially constructed m.s.t.

- the process repeats, adding one edge at a time;
- what cut should we identify? and identify another cut after adding an edge;

# 2. Greedy algorithms

```
function grow-tree(V,E);
     A = empty_set;
     while |A| < |V|-1 do
        if (u,v) not in A
             & is a light edge cross some new cut
             & A U {(u,v)} does not form a cycle
        then A = A U {(u,v)};
     return A;
```

- how to identify a cut (then a light edge)?
- how to check cyclicality?

# 2. Greedy algorithms

This leads to two different MST algorithms: Prim's and Kruskal's

Prim's:

- start from any single vertex $a$, let $S = \{a\}$; $T = \emptyset$;
- find a light edge $(u, v)$ crossing the cut $(S, V - S)$;
  then $T = T \cup \{(u, v)\}$; $S = S \cup \{v\}$;
- if $|T| < n - 1$, repeat the above step;

Technically, how to identify every light edge (efficiently)?

- a new cut evolves from an old cut; a light edge crossing the new cut may be identified with little effort;

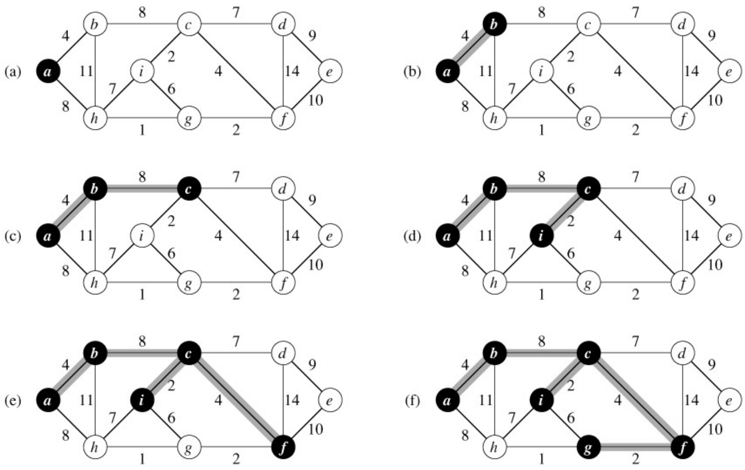
# 2. Greedy algorithms

```
function prim (G, w)
for all u in V
  cost(u) = infinity;
  prev(u) = nil;
pick an arbitrary vertex s
cost(s) = 0;
T = empty_set;
H = makequeue(V);
while H is not empty
  u = dequeue(H);
  T = T U {(prev(u), u)};
   for every (u, v) in E
    if cost(v) > w(u, v)
       cost(v) = w(u, v);
       prev(v) = u;
return (T, prev)
```

# 2. Greedy algorithms

- what does the list `prev` look like?
- does it work on directed graphs?
- what would happen if the graph is not connected?
- how to implement a priority queue?
- time complexity? $O(|E| + |V| \log |V|)$

# 2. Greedy algorithms




dynamic changes of the priority queue.

But wait, are we sure algorithm `prim` finds an m.s.t.?

We need to prove `T` generated by `prim` is an m.s.t.

We prove a more general **claim**:

At every iteration of the while loop, `T` is contained in some m.s.t.
called the **loop-invariant** for the while loop.

We prove the claim by induction on $k$, of the $k^{\text{th}}$ iteration.

# 2. Greedy algorithms

**Claim**: At every iteration of the while loop in algorithm `prim`, set `T` is contained in some m.s.t.

**Proof**:

- base case: $k = 0$,
  the algorithm has yet to enter the while loop. Then $T = \emptyset$, therefore, it is contained in every m.s.t..
- assumption: at iteration $k$, $T \subseteq \mathcal{T}$ for some m.s.t., $\mathcal{T}$.
- induction: at iteration $k+1$, $T' = T \cup \{(u, v)\}$, where edge $(u, v)$ is a light edge cross cut $(S, V - S)$, and $S$ is exactly the set of those vertices in $T$.

  (1) if $(u, v) \in \mathcal{T}$, then $T' \subseteq \mathcal{T}$, we prove the claim.

  (2) otherwise, $\mathcal{T}$ has to contain a different edge $(x, y)$ crossing the cut $(S, V - S)$. (why?)
- let $\mathcal{T}' = \mathcal{T} \cup \{(u, v)\} - \{(x, y)\}$. $\mathcal{T}'$ is also an m.s.t. (why?)
- $T' \subseteq \mathcal{T}'$ (why?), we prove the claim.

# 2. Greedy algorithms

```
function Kruskal (G=(V, E), w)

1. Sort edges by weight in the nondecreasing order;
2. forest F = emptyset;
3. for every edge (u, v) in the sorted order;
4.   if u and v not belonging to the same tree in F
5.      F = F U {(u, v)};
6.      update forest F;
```

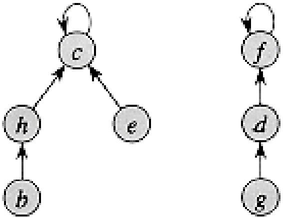
- complexity depends on how to implement steps 2, 4, and 5
- use set to store a tree in F, with operations

*make-setu*, *find*$(u)$, *union*$(u, v)$.

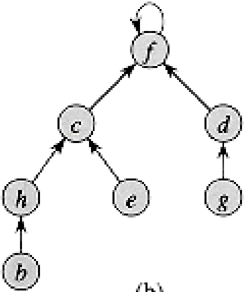
# 2. Greedy algorithms

**Disjoint-set**

- $\textsc{Make Set}(x)$: create a set of single element $x$;
  $\textsc{Find Set}(x)$: identify the set that contains element $x$;
  $\textsc{Union}(x, y)$: union the two sets containing $x$ and $y$ into one;




(a)




(b)

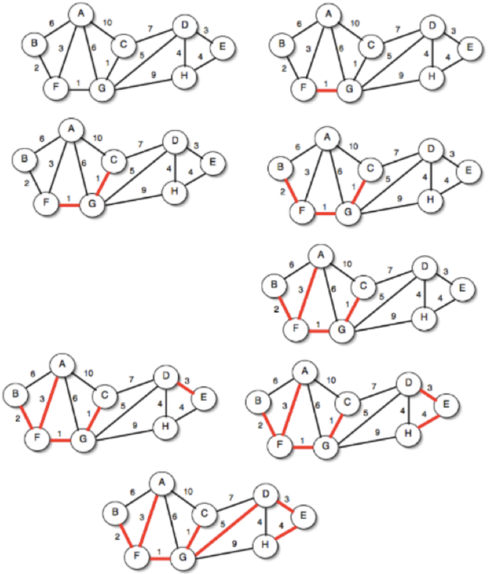
# 2. Greedy algorithms

```
function Kruskal (G=(V, E), w)

Sort edges by weight in the nondecreasing order;
for every u in V,
   make_set(u);
for every edge (u, v) in the sorted order;
  if find(u) not = find(v)
     F = F U {(u, v)};
     union(u, v);
```

Time complexity:

# 2. Greedy algorithms

Execution of `Kruskal`'s:




disjoint sets

[A] [B] [C] [D] [E] [F] [G] [H]

[A] [B] [C] [D] [E] [F] [G] [H]

[A] [B] [C] [D] [E] [F, G] [H]

[A] [B] [D] [E] [C, F, G] [H]

[A] [D] [E] [B, C, F, G] [H]

[D] [E] [A, B, C, F, G] [H]

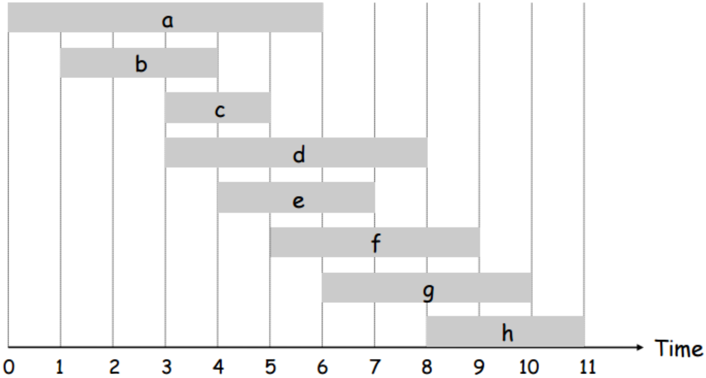
[D, E] [A, B, C, F, G] [H]

[D, E, H] [A, B, C, F, G]

[A, B, C, D, E, F, G, H]

# 2. Greedy algorithms

Problem 3: **Activity Scheduling**
Input: $n$ activities, each with start time $s_i$ and finish time $f_i$;
Output: max number of activities allowed to use a venue exclusively;



# 2. Greedy algorithms

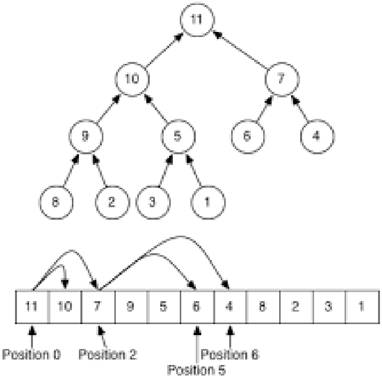
**Greedy-choice property** for **Activity Scheduling**:

The activity with the earliest finish time is contained in some optimal scheduling

Proof: (in-classroom exercise)

# 3. Some data structures and implementations

**heap implementation of priority queue**




- heap: a complete binary tree, in which every node $u$ satisfies:

  for max heap$\texttt{key}(u) \geq \texttt{key}(lc(u))$ and $\texttt{key}(u) \geq \texttt{key}(rc(u))$

- storage: array $A[0..n-1]$, $A[k]$'s children: $A[2k+1]$, $A[2k+2]$;

# 3. Implementations of priority queue and set

function `build-heap`: to build an initial heap

function `heapify`: adjust nodes to satisfy the heap condition

function `increase-key`: update key for a node in the heap

```
function heapify(A, k, n);    // adjust node from position k
                              // and downward
1. if k <= n/2
2.    place in A[k] the largest of A[2k+1], A[2k+2], and A[k]
3.    if index of largest element is not k
4.       k = index of the largest
5.       heapify(A, k, n);
```

# 3. Some data structures and implementations

usage in `prim` and `Dijkstra's` complexity analysis

```
function build-heap(A, n);  // build initial heap

1.   for k = n/2 to 0
2.     heapify(A, k, n)

function increase-key(A, i, key); // update node i's key value

1. if key > A[i]
2      A[i] = key
3. while i > 0 and A[PARENT[i]] < A[i]
4.   exchange A[i] with A[PARENT[i]]
5.   i = PARENT[i]
```

# 4. Matrix multiplication for graphs

$$\underset{2\times 2}{\begin{bmatrix} -2 & 1 \\ 0 & 4 \end{bmatrix}} \times \underset{2\times 2}{\begin{bmatrix} 6 & 5 \\ -7 & 1 \end{bmatrix}} = \underset{2\times 2}{\begin{bmatrix} \boxed{-2\times 6+1\times -7} & -2\times 5+1\times 1 \\ 0\times 6+4\times -7 & 0\times 5+4\times 1 \end{bmatrix}}$$

**dot product**

$$= \begin{bmatrix} \boxed{-19} & -9 \\ -28 & 4 \end{bmatrix}$$

# 4. Matrix multiplication for graphs

$$c_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + a_{14}b_{41}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

2 x 4     4 x 3     2 x 3

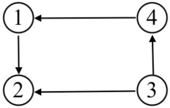
$$c_{22} = a_{21}b_{12} + a_{22}b_{22} + a_{23}b_{32} + a_{24}b_{42}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \\ b_{41} & b_{42} & b_{43} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \end{bmatrix}$$

# 4. Matrix multiplication for graphs

Consider an adjacency matrix of a directed graph:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$




$$A^2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

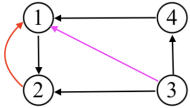
What does $A^2$ mean? e.g., entry $A^2(3,1)$

$$= A(3,1) \times A(1,1) + A(3,2) \times A(2,1) + A(3,3) \times A(3,1) + A(3,4) \times A(4,1)$$
$$= 0 + 0 + 0 + 1 = 1$$

What does $A^2(3,1) = 1$ mean?

# 4. Matrix multiplication for graphs

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$




What should $A^2$ be now?

$$A^2 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 2 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

What does $A^2(3, 1) = 2$ mean?

# 4. Matrix multiplication for graphs

Can we conclude?

If $A_{n \times n}$ is a 0-1 adjacency matrix, then $A^k$ contains the information about the number $k$-step paths $i \rightsquigarrow j$;

- How to get number of paths $i \rightsquigarrow j$, regardless steps?
- What if the given $A(i, i) \neq 0$?
- What if the graph is weighted and shortest paths are desired?

# 5. All pairs shortest paths

All Pair Shortest Paths Problem

Input: A weighted graph $G = (V, E)$ with edge weight function $w$;
Output: Shortest paths between every pair of vertices in $G$.

- If run DIJKSTRA's on every vertex, with total time $O(|V|^2 \log |V| + |V||E|)$, but only on graphs with non-negative edges.
- **Floyd-Warshall** algorithm: $O(|V|^3)$, able to detect negative cycles.

# 4. Matrix multiplication for graphs

## All Pair Shortest Paths

- Can we solve the problem with matrix multiplication?

- Revising dot-product

$$A^2(i,j) = A(i,1) \times A(1,j) + \cdots + A(i,k) \times A(k,j) + \cdots + A(i,n) \times A(n,j)$$

replace $+$ with min;
replace $\times$ with $+$;

- Does the following formulation of shortest distances work?

$$d(i,j) = \min_{1 \le k \le n} \{d(i,k) + d(k,j)\}$$

(Circular data dependencies.)

# 5. All pairs shortest paths

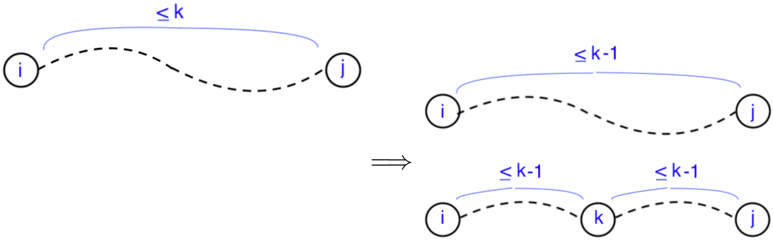
The idea of **Floyd-Warshall** algorithm is to break the "circularity" by computing more refined data.

**Define**: $D^{(k)}[i,j]$ to be the weight of a shortest path between vertices $i$ and $j$ **on which all intermediate nodes are of indexes** $\leq k$.

Note: the goal is still to compute $d_{ij}$, which is $D^{(n)}[i,j]$, where $n = |V|$

# 5. All pairs shortest paths

For $D^{(k)}[i,j]$, we can have recursive formulation, based on two possibilities:

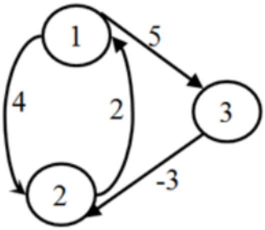



$$D^{(k)}[i,j] = \min \begin{cases} D^{(k-1)}[i,j] & \leftarrow \text{vertex } k \text{ is not on the path} \\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j] & \leftarrow \text{vertex } k \text{ is on the path} \end{cases}$$

base cases: $D^{(0)}[i,j] = w(i,j)$, $D^{(0)} = W$ (there are no intermediate nodes).

# 5. All pairs shortest paths

Example: $W$ is the edge weight matrix;

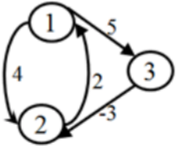



$$W = D^0 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 4 & 5 \\ 2 & 2 & 0 & \infty \\ 3 & \infty & -3 & 0 \end{array}$$

$$P^0 = \begin{array}{c|ccc} & 1 & 2 & 3 \\ \hline 1 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \end{array}$$

$P$ is the $\pi$ paths matrix, storing $k$ values

# 5. All pairs shortest paths




$D^0 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | ∞ |
| 3 | ∞ | -3 | 0 |

k=1: vertex 1 can be intermediate node

$D^1 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | **7** |
| 3 | ∞ | -3 | 0 |

$D^1[2,3] = \min( D^0[2,3], \underline{D^0[2,1]+D^0[1,3]} )$

$= \min (\infty, 7)$

$= \mathbf{7}$

$P^1 =$

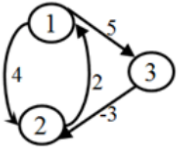
| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 0 | 0 | 0 |

$D^1[3,2] = \min( D^0[3,2], D^0[3,1]+D^0[1,2] )$

$= \min (-3,\infty)$

$= -3$

# 5. All pairs shortest paths




$D^1 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | ∞ | -3 | 0 |

k=2: vertices 1, 2 can be intermediate node

$D^2 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | **-1** | -3 | 0 |

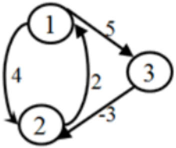
$$D^2[1,3] = \min(D^1[1,3],\ D^1[1,2]+D^1[2,3]) = \min(5,\ 4+7) = 5$$

$P^2 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 2 | 0 | 0 |

$$D^2[3,1] = \min(D^1[3,1],\ \underline{D^1[3,2]+D^1[2,1]}) = \min(\infty,\ -3+2) = \mathbf{-1}$$

# 5. All pairs shortest paths




$D^2 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 4 | 5 |
| 2 | 2 | 0 | 7 |
| 3 | -1 | -3 | 0 |

k=3: vertices 1, 2, 3 can be intermediate node

$D^3 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | **2** | 5 |
| 2 | 2 | 0 | 7 |
| 3 | -1 | -3 | 0 |

$D^3[1,2] = \min(D^2[1,2], \underline{D^2[1,3]+D^2[3,2]})$
$= \min(4, 5+(-3))$
$= \mathbf{2}$

$P^3 =$

| | 1 | 2 | 3 |
|---|---|---|---|
| 1 | 0 | 3 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 2 | 0 | 0 |

$D^3[2,1] = \min(D^2[2,1], D^2[2,3]+D^2[3,1])$
$= \min(2, 7+(-1))$
$= 2$

# 5. All pairs shortest paths

Without paths information

FLOYD-WARSHALL($W$)
1. $n = rows[W]$
2. $D^{(0)} = W$
3. **for** $k = 1$ **to** $n$ $\leftarrow$ for different layer $k$
4. **for** $i = 1$ **to** $n$
5. **for** $j = 1$ **to** $n$ $\swarrow$ compute matrix $D^{(k)}$
6. $D^{(k)}[i,j] = \min \begin{cases} D^{(k-1)}[i,j] \\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \end{cases}$
7. **return** $(D^{(n)})$

Time complexity $O(|V|^3)$.

# 5. All pairs shortest paths

With paths information

initialize path matrices $P = \{P^{(1)}, \ldots, P(^{(n)}\}$ to have zero values

FLOYD-WARSHALL($W$)

$n = rows[W]$<br>
$D^{(0)} = W$<br>
**for** $k = 1$ **to** $n$<br>
  **for** $i = 1$ **to** $n$<br>
    **for** $j = 1$ **to** $n$<br>
      $D^{(k)}[i,j] = \min \begin{cases} D^{(k-1)}[i,j]; \\ D^{(k-1)}[i,k] + D^{(k-1)}[k,j]; \end{cases}$<br>
      **set** $P^{(k)}[i,j] = P^{(k-1)}[i,j]$ **or** $P^{(k)}[i,j] = k$, **accordingly**<br>
**return** $(D^{(n)}, P)$

# Summary of shortest paths algorithms

- A lot of path-related problems can be solved with DFS-like algorithms

  reachability, cycle, path counting, shortest path problems;

  **edge relaxation**: update distance/path based on edge $(u, v)$;
- **single-source shortest paths on DAG**: DAG-Paths-algorithm, DP;
- **single-target shortest paths on DAG**: DAG-Paths-algorithm, DP
- **single-source shortest paths**: Dijkstra's algorithm;
- **single-source shortest paths**: Bellman-Ford algorithm;
- **all-pairs shortest paths**: Floyd-Warshall algorithm, DP;

## 6. Linear programming and Max Flow

Example-1: **Knapsack** can be written as

Find $(x_1, x_2, \ldots, x_n)$, such that

$$x_1 v_1 + x_2 v_2 + \cdots + x_n v_n = \sum_{k=1}^{n} x_i v_i \text{ is maximized}$$

subject to

$$x_1 s_1 + \ldots x_n s_n = \sum_{k=1}^{n} x_i s_i \leq B$$

$$x_i \in \{0, 1\}$$

# 6. Linear programming and Max Flow

Example-2: **MST** can be written as

Find $(e_1, x_2, \ldots, e_m)$, such that

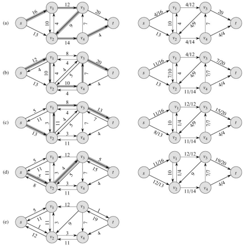
$$e_1 w_1 + e_2 w_2 + \cdots + e_m w_m = \sum_{k=1}^{m} e_i w_i \text{ is minimized}$$

subject to

$$e_1 + \ldots e_m = \sum_{k=1}^{m} e_i = n - 1$$
$$e_i \in \{0, 1\},\ 1 \leq i \leq m$$
$$\sum_{k_i} e_{k_i} \geq 1, \text{ where } e_{k_i} \text{ incident on vertex } k,\ 1 \leq k \leq n$$

# 6. Linear programming and Max Flow

Example-3 **Max Flow**:



Find $(f_1, f_2, \ldots, f_m)$, such that

$$\sum_j f_{s_j} \text{ is maximized}$$

where $e_{s_j}$ are outgoing edges from source $s$,

subject to

$f_i \leq w(e_i)$, $1 \leq i \leq m$

$\sum_i f_{i_k} = \sum_j f_{k_j}$, $1 \leq k \leq n$

# 6. Linear programming and Max Flow

**General linear program format**:

$$\max \mathbf{c}^T\mathbf{x} \quad \text{or} \quad \min \mathbf{c}^T\mathbf{x}$$

subject to

$$\mathbf{A}\mathbf{x} \leq \mathbf{b} \text{ or } \quad \geq \mathbf{b}$$

where

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \dots \\ x_n \end{pmatrix} \quad \mathbf{c} = \begin{pmatrix} c_1 \\ c_2 \\ \dots \\ c_n \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \dots \\ b_m \end{pmatrix} \quad \mathbf{A} = \begin{pmatrix} a_{1,1} & \dots & a_{1,n} \\ a_{2,1} & \dots & a_{2,n} \\ \dots & & \\ a_{m,1} & \dots & a_{m,n} \end{pmatrix} \tag{1}$$