Programming Design In-class Practices

Complexity and Graphs

Ling-Chieh Kung

Department of Information Management
National Taiwan University

Problem 1: Big-O practices

- Determine whether the following big-O relationships are valid or not:
  - $100n \in O(n)$.
  - $100n + 5 \in O(n)$.
  - $100n + m^2 \in O(n + m^2)$.
  - $n^2 m + n \log n \in O(n^2 m)$.
  - $2^n \in O(n!)$.
  - $n^2 \in O(2^n)$.

Problem 2: Big-O practices

- For each of the following functions, find a valid $g(n)$ or $g(n, m)$ to validate the big-O relationship. Let your answer be as small as possible.
  - $\sqrt{n} + n \in O(g(n))$.
  - $mn \log n + m^2 n \in O(g(n, m))$.
  - $100n + m^2 + n! \in O(g(n, m))$.
  - $n(m + n \log n) \in O(g(n, m))$.
  - $2^{n+2} m \in O(g(n, m))$.
  - $1 + 3 + 5 + 7 + \cdots + (2n - 1) \in O(g(n))$.
  - $1 \times 1 + 3 \times 2 + 5 \times 4 + 7 \times 8 + \cdots + (2n - 1)2^{n-1} \in O(g(n))$.

Problem 3: bubble vs. insertion

- Consider bubble sort and insertion sort (implemented in the next page).
  - Find the complexity for each algorithm.
  - Determine whether their execution time will be different in average.

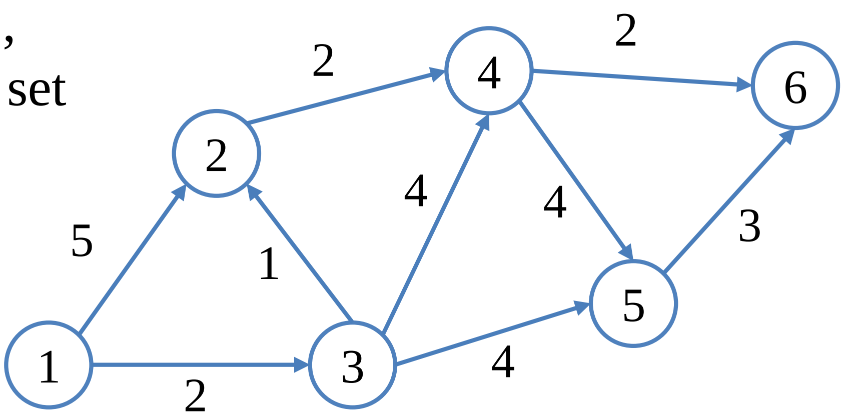
Problem 3: bubble vs. insertion

```
void bubbleSort(const int unsorted[],
                int sorted[], int len)
{
  for(int i = 0; i < len; i++)
    sorted[i] = unsorted[i];

  for(int i = len - 1; i > 0; i--) {
    for(int j = 0; j < i; j++) {
      if(sorted[j] > sorted[j + 1]) {
        int temp = sorted[j];
        sorted[j] = sorted[j + 1];
        sorted[j + 1] = temp;
      }
    }
  }
}
```

```
void insertionSort(const int unsorted[],
                   int sorted[], int len)
{
  for(int i = 0; i < len; i++)
    sorted[i] = unsorted[i];

  for(int i = 0; i < len; i++) {
    for(int j = i; j > 0; j--) {
      if(sorted[j] < sorted[j - 1]) {
        int temp = sorted[j];
        sorted[j] = sorted[j - 1];
        sorted[j - 1] = temp;
      }
      else
        break;
    }
  }
}
```

Shortest paths

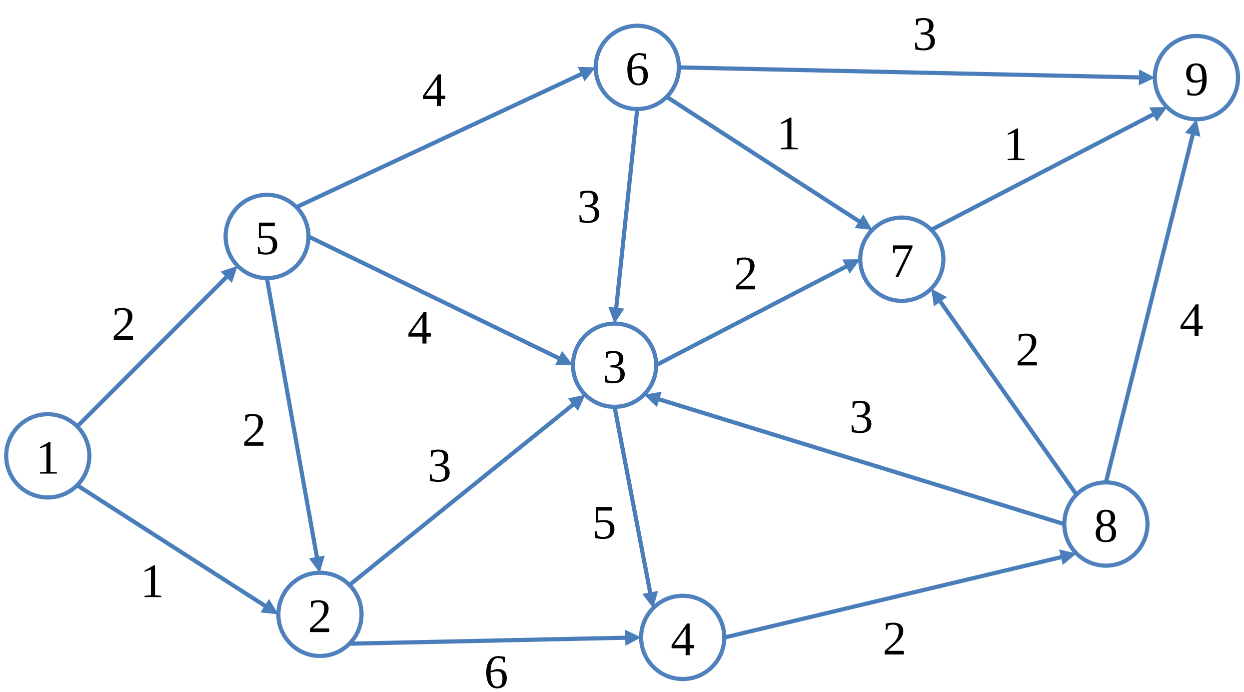
- We are given a directed graph $G = (V, A)$, where V is the set of vertices and A is the set of arcs.
  - d_{uv} is the distance of arc $(u, v) \in A$.
- We want to find the **shortest path** from vertex $s \in V$ to vertex $t \in V$.
- Assume that G is **acyclic**.



- In this example:
  - From 1 to 6: (1, 3, 2, 4, 6), distance = 7.
  - From 1 to 5: (1, 3, 5), distance = 6.
  - From 2 to 6: (2, 4, 6), distance = 4.

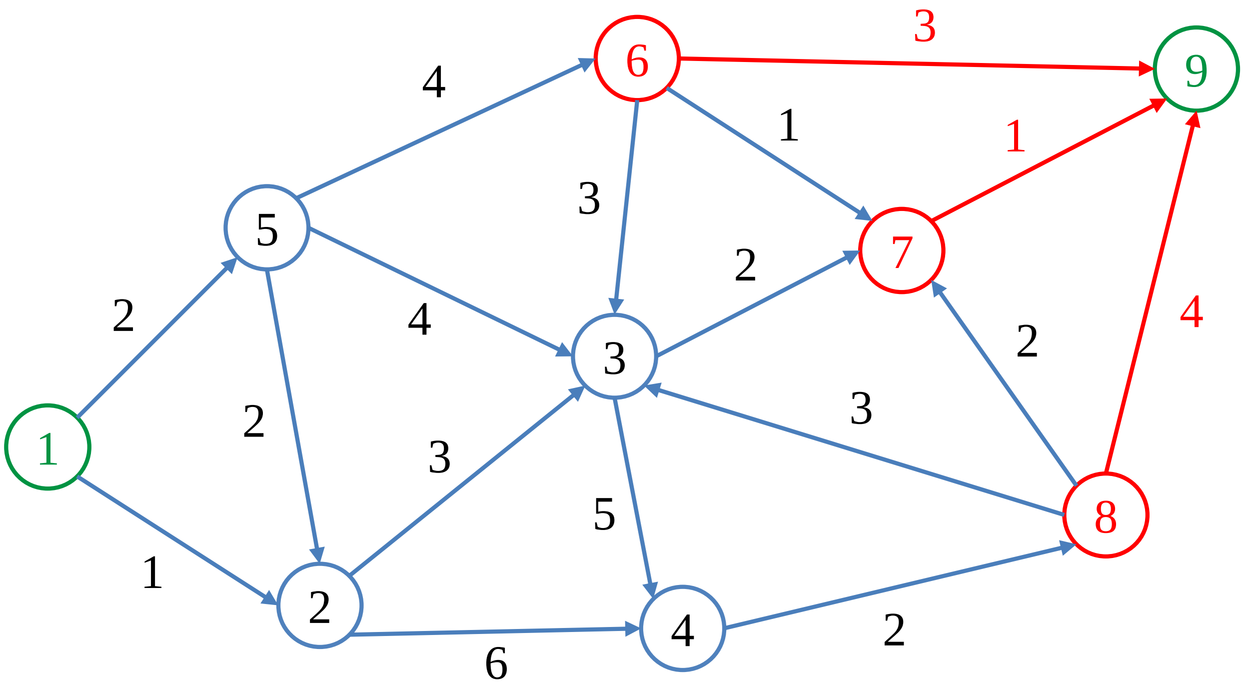
Shortest paths

- How to find the shortest path from s to t in general?



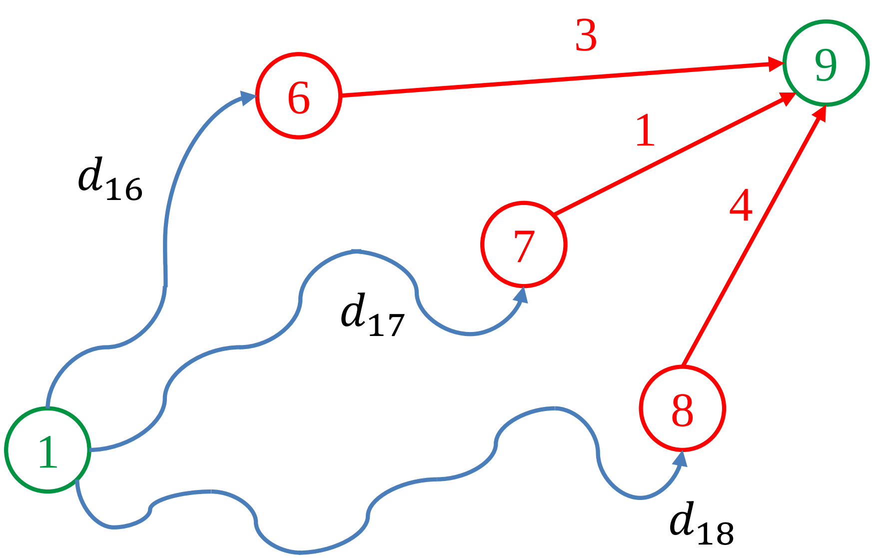
Shortest paths: idea

- Let's go from $s = 1$ to $t = 9$.
  - Somehow we need to get to at least one of t's **incoming neighbor**.



Shortest paths: idea

- If we know how to get to each incoming neighbor of t in the best way, we know how to get to t in the best way.
  - It is the smallest among $d_{16} + 3$, $d_{17} + 1$, and $d_{18} + 4$.
  - How to find d_{16}? **Recursion**!



Shortest paths: idea

- More precisely, let $N^I(t)$ be the set of incoming neighbors of t.
- Let d_{uv} be the distance of the shortest path from $u \in V$ to $v \in V$.
- We have

$$d_{st} = \min_{v \in N^I(t)} \{d_{sv} + d_{vt}\}.$$

- To find d_{sv}, treat v as the destination and play the same trick.

Problem 4: pseudocode

- Please write **pseudocode** of the above problem solving strategy for the shortest path problem.

Problem 5: implementation

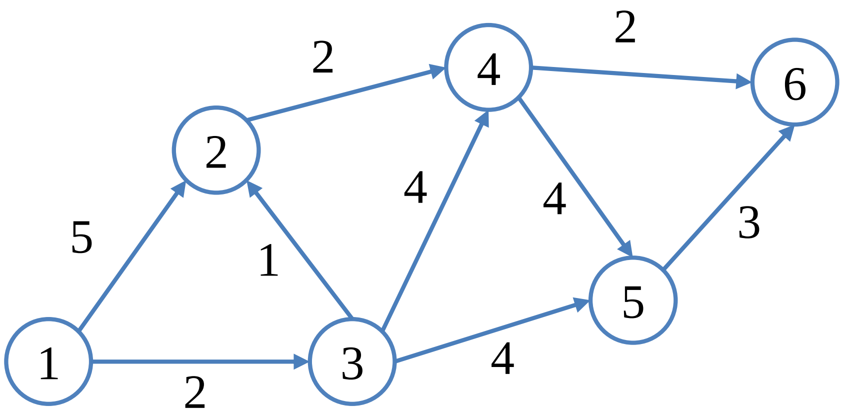
- Given a graph and a pair of source and destination vertices, find a shortest path.
- Let n be the number of vertices, m be the number of arcs, $V = \{1, \dots, n\}$ be the set of vertices, $A = \{(u, v) | u \in V, v \in V\}$ be the set of arcs, $s \in V$ be the ID of the source vertex, and $t \in V$ be the ID of destination set.
- Input:
  - The first line: $n \in V$, $m \in \{1, \dots, n(n-1)\}$, $s \in V$, and $t \in V$.
  - The ith line, $i = 2, \dots, n+1$: $d_{i-1,1}$, $d_{i-1,2}$, …, and d_{in}.
  - $d_{uv} = -1$ if $u = v$ or $(u, v) \notin A$; d_{uv} is the distance of (u, v) otherwise.
  - Two consecutive values in one line are separated by a white space.
  - $n \leq 10$, $m \leq n(n-1)$, $d_{uv} \in \{1, \dots, 100\}$.
- Output:
  - The distance of any shortest path from s to t.

Problem 5: implementation

- Sample Input/output:

Input:
```
6 9 1 6
-1 5 2 -1 -1 -1
-1 -1 -1 2 -1 -1
-1 1 -1 4 4 -1
-1 -1 -1 -1 4 2
-1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1
```

Output:
```
7
```

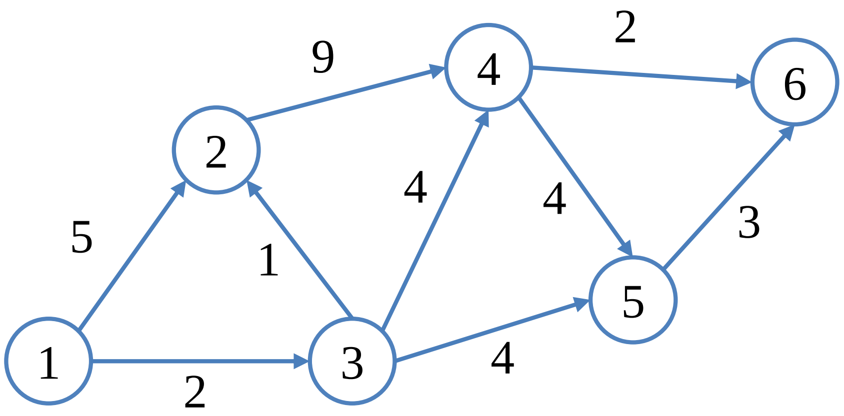


Problem 5: implementation

- Sample Input/output:

Input:
```
6 9 1 6
-1 5 2 -1 -1 -1
-1 -1 -1 9 -1 -1
-1 1 -1 4 4 -1
-1 -1 -1 -1 4 2
-1 -1 -1 -1 -1 3
-1 -1 -1 -1 -1 -1
```

Output:
```
8
```



Weakness of recursion

- The above naïve recursion works for small-scale instances only.
- What if the instance size is big (e.g., $n = 100$ and $m = 4000$)?
- We may do "smart" recursion:
  - Do not solve a subproblem twice.
  - For function parameters, add **one additional array** to record the shortest path distances for all **solved** subproblems.
- In the function, check that array to see whether we have the solution already.
  - If so, return it directly.
  - Otherwise, solve it recursively.