Chapter 1

GRAPH ALGORITHMS

- **Definitions**

  directed
  complete
  weight
  subgraph
  path
  independent set
  cycle
  clique
  tree, forest
  bipartite
  connected
  colouring
  P problems
  NP problems, NPC
  adjacency matrix
  sparse $|E| \ll |V^2|/2$
  dense $|E| \sim |V^2|/2$
  MATROIDS
  GREEDY algorithms

$$A = \begin{array}{|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 0 & \circ\ \circ\ \circ & \\ \hline 1 & 0 & 1 & 0 & \circ\ \circ\ \circ & \\ \hline 1 & 1 & 0 & 1 & \circ\ \circ\ \circ & \\ \hline \circ & \circ & \circ & \circ & & \\ \circ & \circ & \circ & \circ & & \\ \circ & \circ & \circ & \circ & & \\ & & & & \circ\ \circ\ \circ & 0 \\ \hline \end{array}$$

Representations

  – matrix representation

- linked list

Basic Graph Problems

Application in the practice.

Many different issues.

- **Small Overview**

  "Easy" problems [Solvable by Polynomial Algorithm]

  connectivity problems
  shortest (weighted) path
  minimum spanning tree
  Flow problems
  Bipartite graph matching
  minimum cut
  etc.

  "Hard" problems [Non Solvable in P, belongs to NP or NPC]

  graph isomorphy
  max. independent set
  colouring
  Hamiltonian path
  max. clique
  max. cut.
  etc.

Matrix Techniques

The number of the paths leading from v_i to v_j and length ≤ 2.

- **Operations:** $\cdot$; $+$

  The $A^r = a_{i,j}^{(r)}$ tells how many path leads from v_i to v_j which length $\leq r$.

- **Operations: Bool AND; Bool OR**

  The $C^r = c_{i,j}^{(r)}$ tells wether is there any path leading from v_i to v_j having length $\leq r$.

Transitive Closure of a Graph

- **Definition** $G = (V, E)$ graph, $G^* = (V, E^*)$ is its transitive closure if

$$(v_i, v_j) \in E^* \Longleftrightarrow \exists \text{ path in } G \text{ from } v_i \text{ to } v_j.$$

- **Computing the C closure matrix**

  We have to compute the $A^{(n-2)}$.

  Instead of computing $A^1, A^2, A^3, \ldots, A^{n-2}$ we multiply $A^r \times A^r$, so $A^1, A^2, A^4, \ldots, A^{n-1}$.

  We need: $\lceil \log(n-1) \rceil$ matrix multiplication. (Any matrix multiplication method can be used.)

  Sequential complexity: $\Theta(n^3)$.

  Transitive closure sequential complexity:

$$\Theta(n^3 \cdot \log n)$$

- **Parallel formulation**

  E.g.: DNS matrix multiplication on hypercube n^3 processors, $\Theta(\log n)$ parallel execution time.

  Transitive closure parallel execution time:

$$\Theta(\log^2 n)$$

Single Source Shortest Path

We want to get
the minimum weight path
to each $U \in V - \{v\}$
where $G = (V, E, w)$

(Why we do not start with a single $(v : u)$ pair?)

- **Dijkstra algorithm** (G directed, $w(\cdot\ ,\cdot) \geq 0$) **GREEDY!**

V_T the set we investigate,
$l[]$ array stores the weights.
to each $U \in V - \{v\}$
where $G = (V, E, w)$

  - $V_T := \{s\}$
  - Initialize $l[]$:

$$l[v] = \begin{cases} w(s,v) & (s,v) \in E \\ \infty & \text{otherw.} \end{cases}$$

  - Iteration: odd new vertex to V_T while $V_T \neq V$
    take $u \in V \backslash V_T$ where $l[u]$ is the minimum
    $V_T = \{u\} \bigcup V_T$
    recompute the weight array

$$l[v] = \min\{l[v], l[u] + w(u,v)\}$$
$$(V \notin V_T)$$

- **Sequential complexity:** $\Theta(n^2)$
- **Parallel formulation**

  We cannot add more than one vertex in one step. Parallelism
  At each iteration step:
  can take place only in computing the weights. The block striped representation of the A matrix:

- Every P_i processor selects the min. weight node $\rightarrow$ 'single accumulation' P_1 gets the global minimum.
- P_1 broadcast the selected element (e.g. 'u'') and its weight $l[u]$. P_i responsible for u marks $u \in V_T$.
- P_i processors computes the weights

$$l[v] = \min\{l[v],\ l[u]\ +\ w(u,v)\}$$

in the local array — by broadcasts — in block stripe of A

- **Mapping onto different architectures**

Hypercube

- Each processor computes the min:

$$\Theta(\frac{n}{p})$$

- Single node accumulation and one to all broadcast:

$$\Theta(\log p)$$

- Parallel running time:

$$\underbrace{\Theta(n \cdot \frac{n}{p})}_{\text{computation}} + \underbrace{\Theta(n \cdot \log p)}_{\text{communication}}$$

Mesh:

- Computing factor inside the processors are the same.
- Single node accumulation + broadcast:

$$\Theta(\sqrt{p})$$

- Parallel running time:

$$\underbrace{\Theta(n \cdot \frac{n}{p})}_{\text{computation}} + \underbrace{\Theta(n \cdot \sqrt{p})}_{\text{communication}}$$

All Pairs Shortest Path

We want to compute the minimum path to each pair of vertices. The results is the D matrix which contains the weights.

- **Dijkstra**

  We execute the single source shortest path for every $v \in V$ vertices.

  The single sequential complexity:

  $$\Theta(n^2)$$

  The all pairs shortest path complexity:

  $$\Theta(n^3)$$

- **Parallel formulation**

  We execute the single source s.p. sequentally for every vertex on a separate processor.

  We use n processor, the parallel time:

  $$\Theta(n^2)$$

  There is no communication between the processors, no overhead, the efficiency is 1.

  We execute every single source shortest path on a set of processors. In this manner we can use up to n^2 processors.

- **Mesh architecture**

  We have p processor.

  A specific single source shortest path is executed by

  $$\sqrt{\frac{p}{n}} \times \sqrt{\frac{p}{n}} = \frac{p}{n}$$

  processors.

  No communication between the processor blocks.

  Parallel running time:

  $$\Theta(n \cdot \frac{n}{\frac{p}{n}}) + \Theta(n \cdot \sqrt{\frac{p}{n}}) = \Theta(\frac{n^3}{p}) + \Theta(\sqrt{n \cdot p})$$

- **Compairing the two parallel formulation**

  - A) n Processor $\Theta(n^2)$ time
  - B) $p = n^{1.66}$ processor $\Theta(n^{1.33})$ time. Better scalability.

Matrix Algorithm

- **The only assumption:** no neagtive weight cycles.
- **The basic idea:** Let $p_{i,j}^{(k)}$ the minimum weight path from v_i to v_j which has at most k edges and its weight $d_{i,j}^{(k)}$.
  If v_m on the $p_{i,j}^{(k)}$ path we can decompose it into two subpath: $p_{i,m}^{(k-1)} + p_{m,j}^{(1)}$
  Then
  $$d_{i,j}^{(k)} = min_m\{d_{i,j}^{(k-1)}, min_m\{d_{i,m}^{(k-1)} + w(m,j)\}\}$$
  or simply
  $$d_{i,j}^{(k)} = min_m\{d_{i,m}^{(k-1)} + w(m,j)\}$$
- **Start:** $D^{(1)} = \{d_{i,j}^{1}\} = \{a_{i,j}\} = A$
- **Aim:** $D^{(n-1)}$ because the shortest path cannot be longer than $(n-1)$.
- **Computation** Operations:
  multiplication $\Longrightarrow$ addition
  $d_{i,j}^{(k)} = min_m\{d_{i,m}^{(k-1)} + w(m,j)\}$
  addition $\Longrightarrow$ minimum$(\cdot)$ function

  We do not compute $D^{(1)}, D^{(2)}, D^{(3)}, \ldots D^{(n-1)}$ but $D^{(1)}, D^{(2)}, D^{(4)}, \ldots D^{(n-1)}$ by multiplying the matrices by itself. We need $\lceil log(n-1) \rceil$ matrix multiplication.
- **Complexity**

  Sequential complexity of matrix multiplication: $\Theta(n^3)$

  Sequential complexity of this algorithm: $\Theta(n^3 \cdot logn)$

  (Recall Dijkstra has only $\Theta(n^3)$!)

  Using a DNS matrix multiplication algorithm on n^3 processor, the parallel runtime: $\Theta(log^2 n)$

Floyd Algorithm

Not the fastest, but $\Theta(n)$ time with $\Theta(n^2)$ processor, shows useful mapping techniques.

- **Basic idea:** restrict the attention into a $V_k \subseteq V$ subset, and extend it into V.
  $p_{i,j}^{(k)}$ denotes the minimum weight path from v_i to v_j where the internal point are in V_k.
- **Start:** $V_1 = \{v_1\}$
- **Step:** $V_k = V_{k-1} \cup \{v_k\}$
  We have to recompute the path's weights $d_{i,j}^{(k)}$
  - if $p_{i,j}^{(k)} \not\ni v_k$ then $d_{i,j}^{(k)} = d_{i,j}^{(k-1)}$
  - if $p_{i,j}^{(k)} \ni v_k$ then split into two paths $p_{i,k}^{(k-1)}$ $p_{k,j}^{(k-1)}$ (They are in V_k !)

$$d_{i,j}^{(k)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$$

- **Recurrence relation**

$$d_{i,j}^{(k)} = \begin{cases} w(i,j) \text{ if } k = 0 \\ min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} \quad k \geq 1 \end{cases}$$

- **Aim:** $D^{(n-1)}$ matrix.
- **The algorithm:**

$$LOOP_{k=1,\dots n}\left[LOOP_{i=1,\dots n}\left[LOOP_{j=1,\dots n}\left[d_{i,j}^{(k)} = min\{\cdot\}\right]\right]\right]$$

- Sequential time: $\Theta(n^3)$
- **Parallel Formulation:** Block checkerboard mapping, one processor maintain $\frac{n}{\sqrt{p}} \times \frac{n}{\sqrt{p}}$ submatrix.
  (This is only logical mapping!)

- **How it works**

  The k-th step every processor computes

  $$d_{i,j}^{(k)} = min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$$

  it is here — in the same row — in the same column

  Before executing the k-th step the processor maintaining the fragment of the k-th row or column sends the value to the neighbours.

- **Computing steps:**

  - broadcasts k row and column
  - wait till every process has it
  - compute $d_{i,j}^{(k)}$

- **Hypercube architecture** Parallel runtime:

  $$\Theta(n \cdot \frac{n^2}{p}) + \Theta(n \cdot \frac{n}{\sqrt{p}} \cdot logp)$$

- **Bottleneck:** Synchronisation ! The algorithm remains correct if we drop the synchronisation: every processor computes immediately when it gets the $d_{i,j}^{(k)}$ values and then sends to its neighbours.

  Pipelined block checkerboarding.

  The last processor finishes:

  - it gets the first value after $\frac{n}{\sqrt{p}} \cdot \sqrt{p} = \Theta(n)$ time.
  - then it computes $\Theta(\frac{n^3}{p})$

  The parallel runtime:

  $$\Theta(\frac{n^3}{p}) + \Theta(n)$$

If we use $p = n^2$ processors :

$$\Theta(n)$$