A Supernodal All-Pairs Shortest Path Algorithm

Paper by Piyush Sao, Ramakrishnan Kannan, Prasun Gera, Richard Vuduc

Slides by Hyunmo Sung,
Yonsei university Muti-core programming topics Seminar,
Spring 2020

Contents

Introduction

Notations and classic algorithms

Background

Classic Floyd-Warshall algorithm, Min-Plus matrix multiplication, etc..

Supernodal Floyd-Warshall

The method which this paper provide

Asymptotic analysis

The numeric analysis of speed-up and efficiency.

Conclusion

The experiments and results..

Introduction

Graph notation

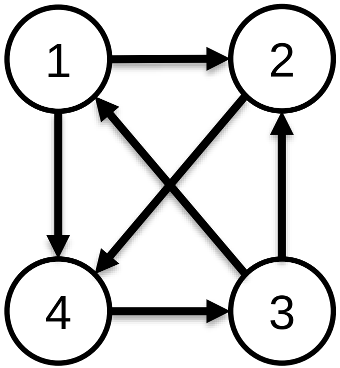


Figure 1. Directed graph(Digraph)

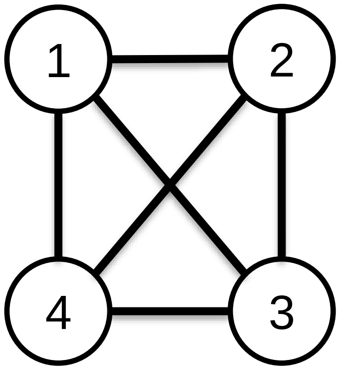


Figure 2.Undirected graph

Graph G defined as $G = (V, E)$.
Which V is a set of vertices and $E \subset V \times V$.

Introduction

Graph notation

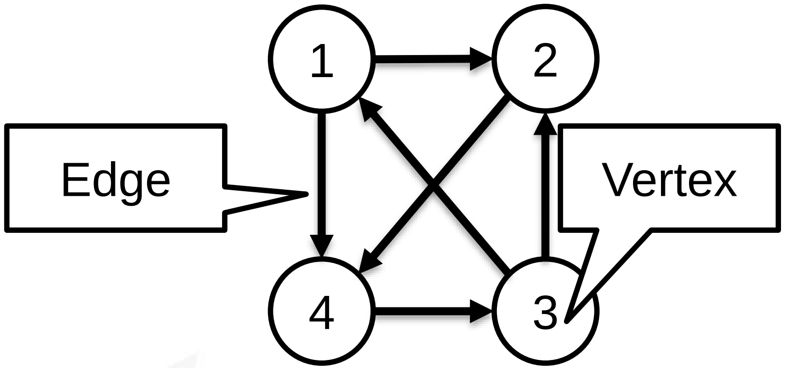


Figure 1. Directed graph(Digraph)

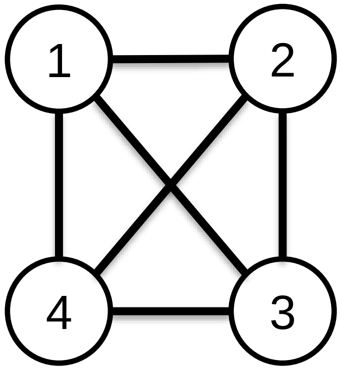
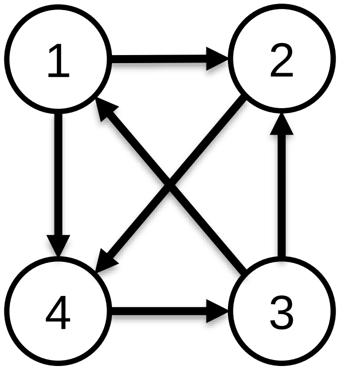


Figure 2.Undirected graph

Graph G defined as $G = (V, E)$.
Which V is a set of vertices and $E \subset V \times V$.

Introduction

Graph notation



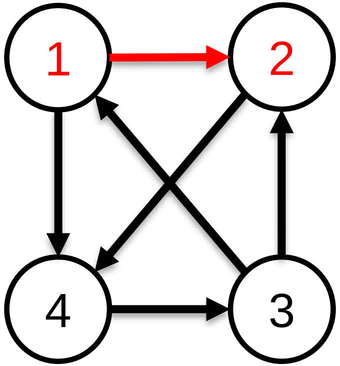
	1	2	3	4
1	X	O	X	O
2	X	X	X	O
3	O	O	X	X
4	X	X	O	X

Figure 3. Directed graph(Digraph) and corresponding adjacency matrix

Graph can be represented by an adjacency matrix

Introduction

Graph notation



To

From

	1	2	3	4
1	X	O	X	O
2	X	X	X	O
3	O	O	X	X
4	X	X	O	X

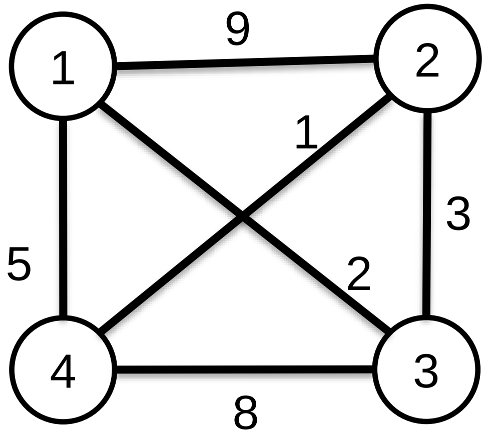
Exists

Figure 3. Directed graph(Digraph) and corresponding adjacency matrix

Graph can be represented by an adjacency matrix

Introduction

Graph notation



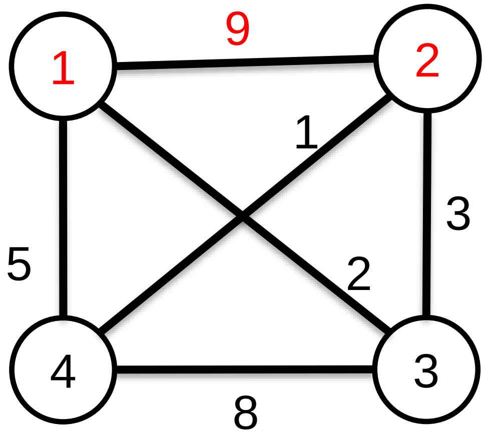
	1	**2**	**3**	**4**
1	X	9	2	5
2	9	X	3	1
3	2	3	X	8
4	5	1	8	X

Figure 4. Weighted undirected graph

Some graphs' edges have weights.
It's used for cost between two points.

Introduction

Graph notation



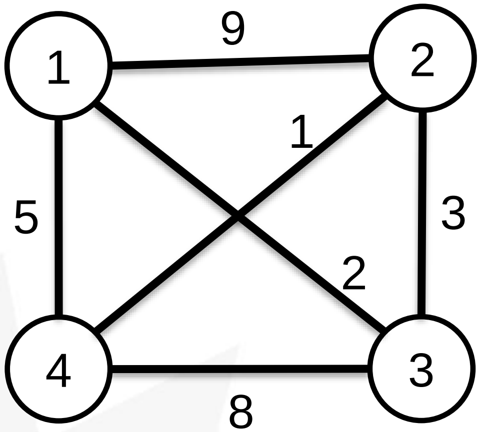
	1	2	3	4
1	X	9	2	5
2	9	X	3	1
3	2	3	X	8
4	5	1	8	X

Figure 4. Weighted undirected graph

Some graphs' edges have weights.
It's used for cost between two points.

Introduction

What is all pairs shortest paths(APSP) problem



	1	2	3	4
1	X	9	2	5
2	9	X	3	1
3	2	3	X	8
4	5	1	8	X

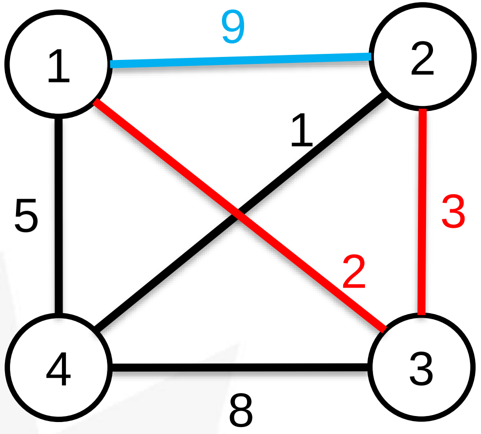
	1	2	3	4
1	4	5	2	5
2	5	2	3	1
3	2	3	4	4
4	5	1	4	2

Figure 5. Weighted undirected graph and its APSP solution

All pairs shortest path problem is a problem to solve the smallest path cost between all pairs.
Example is above.

Introduction

What is all pairs shortest paths(APSP) problem



	1	2	3	4
1	X	9	2	5
2	9	X	3	1
3	2	3	X	8
4	5	1	8	X

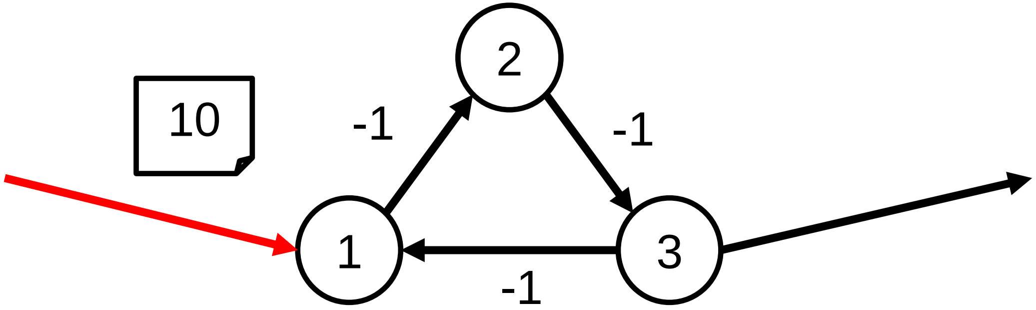
	1	2	3	4
1	4	5	2	5
2	5	2	3	1
3	2	3	4	4
4	5	1	4	2

Figure 5. Weighted undirected graph and its APSP solution

All pairs shortest path problem is a problem to solve the smallest path cost between all pairs.
Example is above.

Introduction

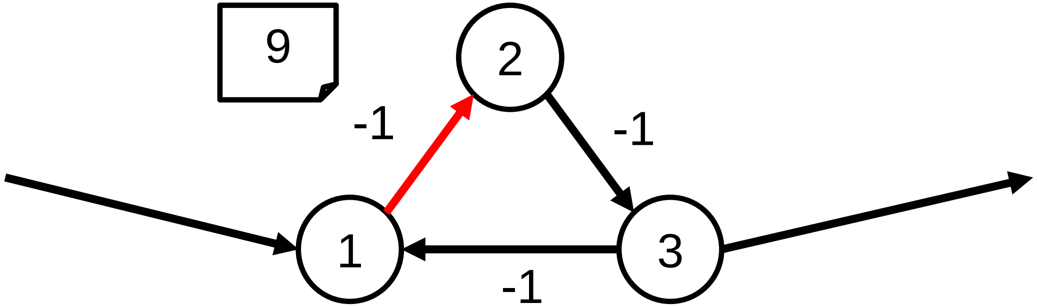
What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

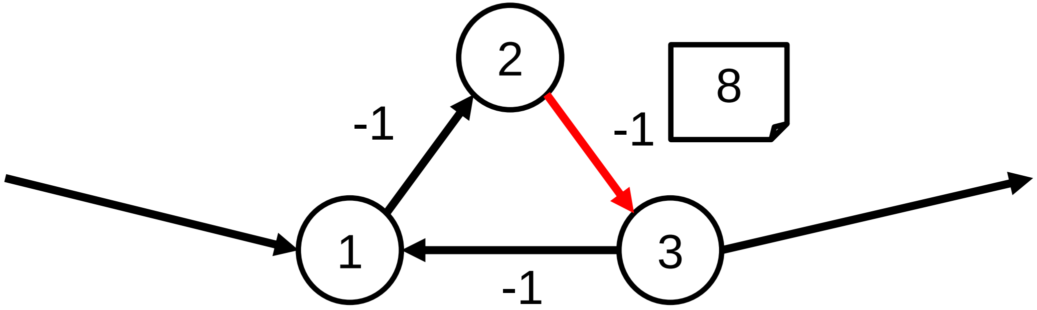
What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

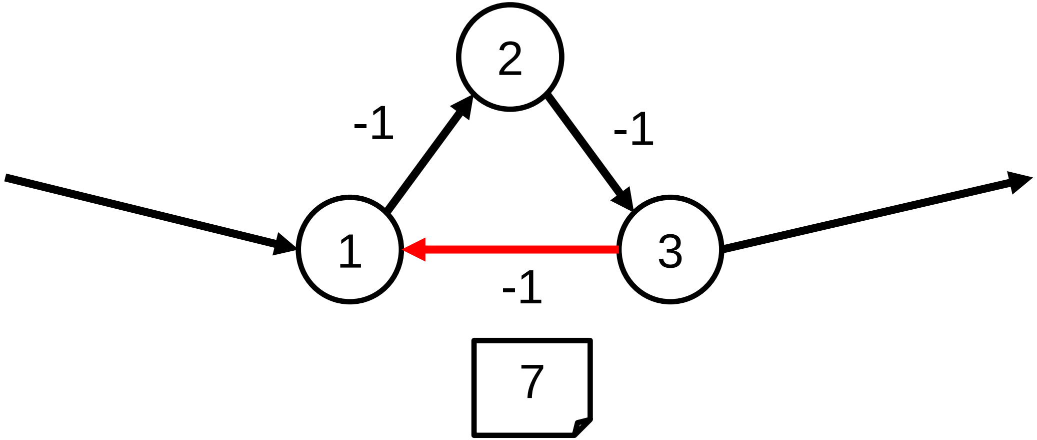
What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

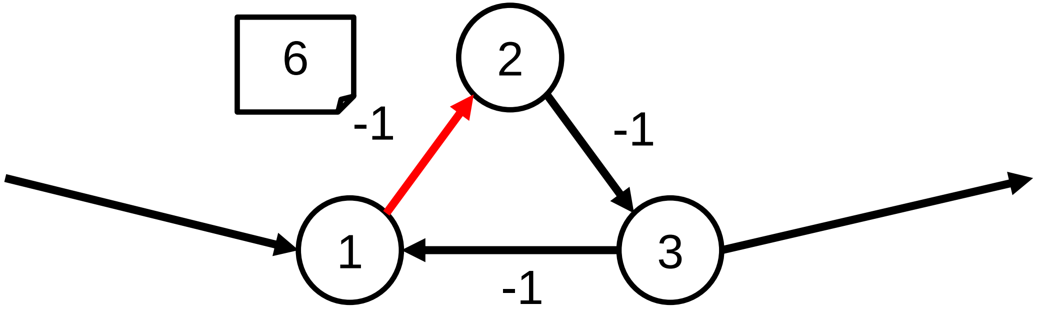
What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

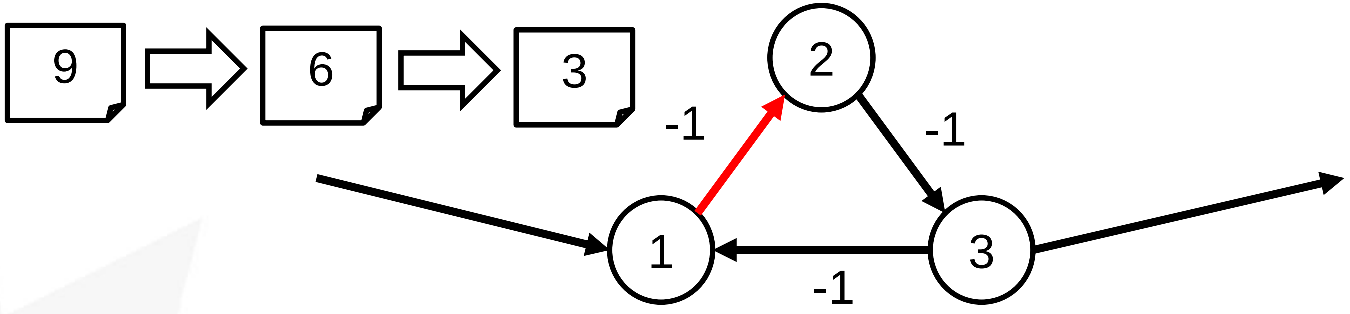
What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

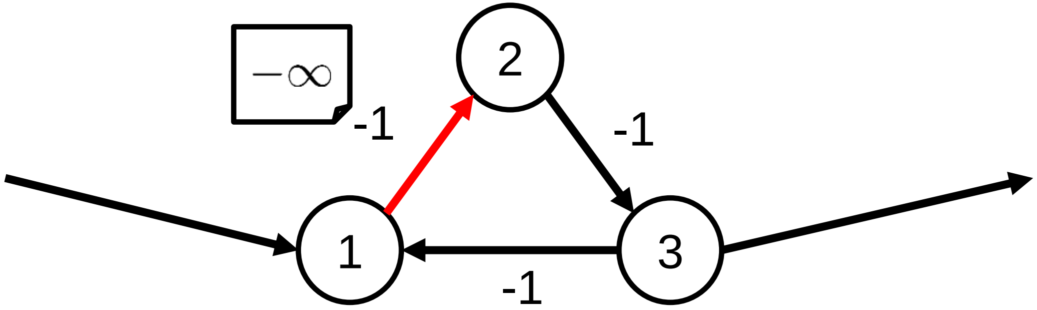
What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

What is all pairs shortest paths(APSP) problem



There is no negative cycle at APSP.
Otherwise, problem will be violated.

Introduction

Target graph

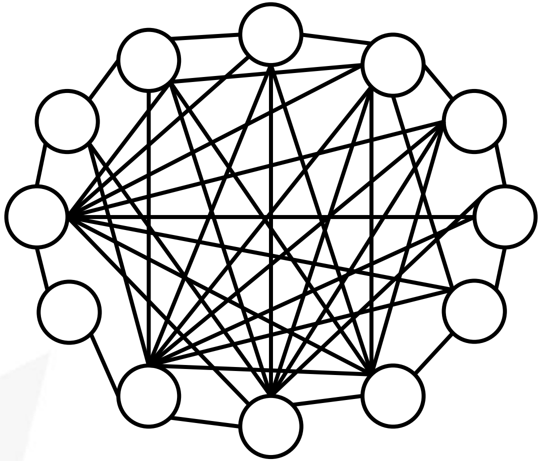


Figure 6. Dense graph

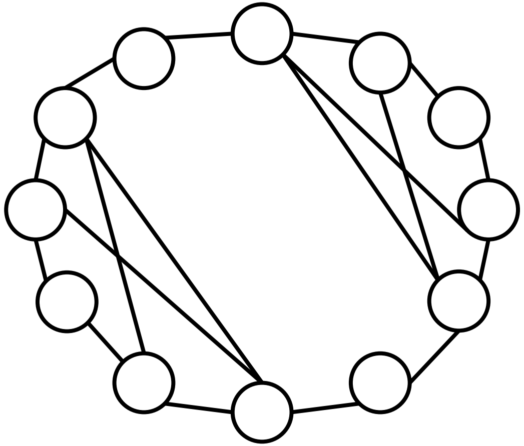


Figure 7. Sparse graph

Graphs without many edges are called sparse graph.
This paper's target is a weighted bisectable sparse undirected graph.
It assumed there is no negative edges.

Introduction

Target graph

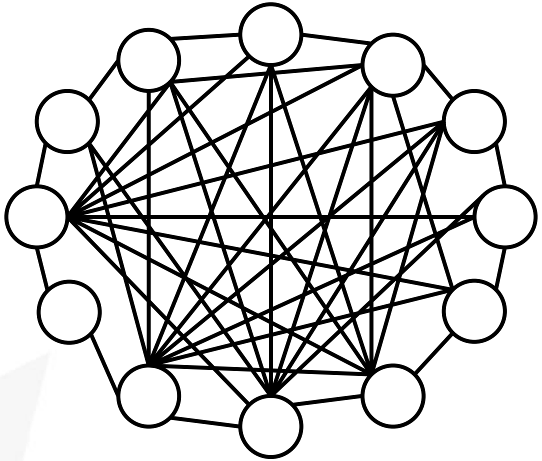


Figure 6. Dense graph

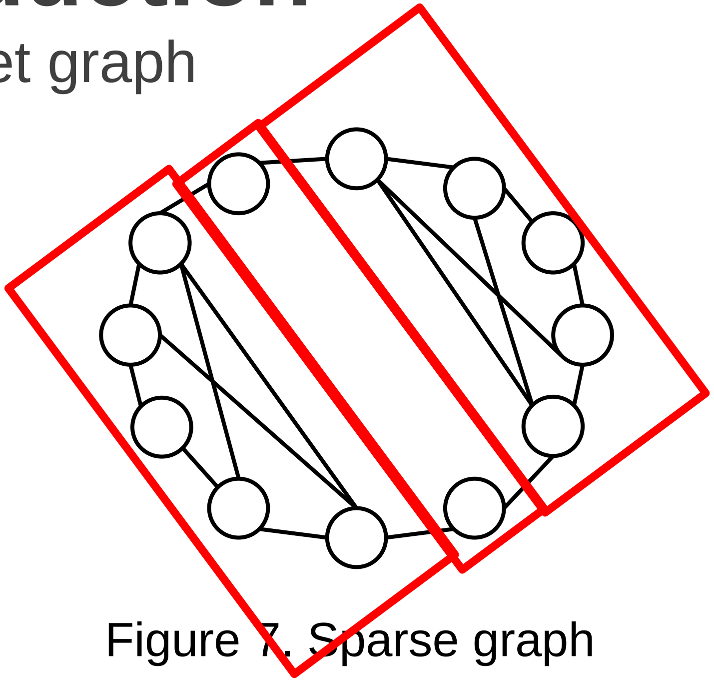


Figure 7. Sparse graph

Graphs without many edges are called sparse graph.
This paper's target is a weighted bisectable sparse undirected graph.
It assumed there is no negative edges.

Introduction

Target graph

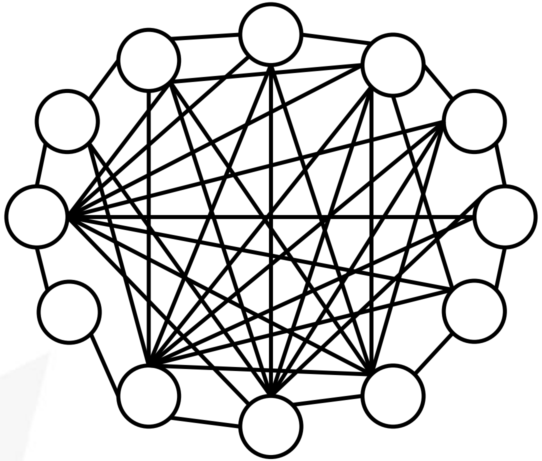


Figure 6. Dense graph

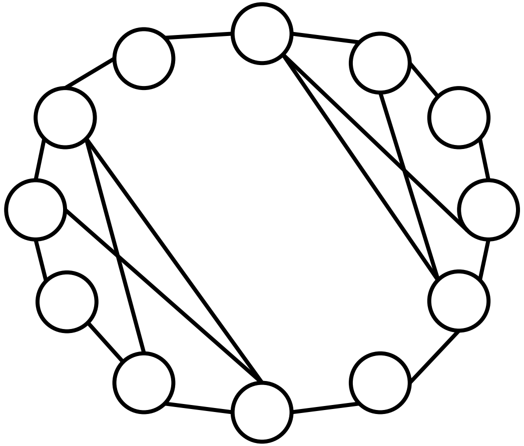
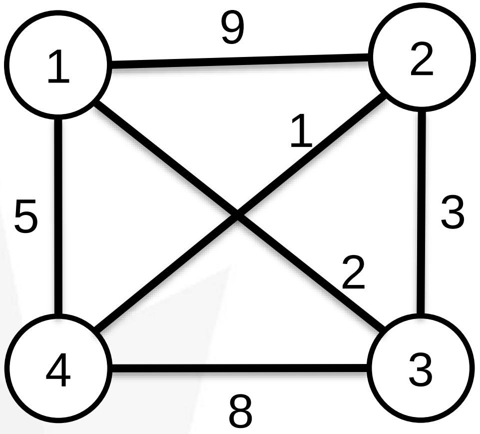


Figure 7. Sparse graph

However, it can be extended to directed graphs with negative edges since it is based on a Floyd-warshall algorithm.

Introduction

Approach from SINGLE source path problem(SSSP)



	1	**2**	**3**	**4**
1	X	9	2	5
2	9	X	3	1
3	2	3	X	8
4	5	1	8	X

	1	**2**	**3**	**4**
1	4	5	2	5

Figure 8. Weighted undirected graph and its SSSP solution

Traditional approach is a solving a SINGLE source path problem many times.
Two major algorithms for SSSP are a Dijkstra's algorithm and a Bellman-ford algorithm.

Introduction

Dijkstra's algorithm

```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

Figure 9. Dijkstra's algorithm

Operation	Complexity
Find-min	$\Theta(1)$
Delete-min	$O(logn)$
Insert	$\Theta(1)$
Decrease-key	$\Theta(1)$
Meld	$\Theta(1)$

Figure 10. Fibonacci heap's compexity

Dijkstra's algorithm is an algorithm that solves a SINGLE source shortest path problem.
If it uses a Fibonacci heap, it takes $O(|V|log|V| + |E|)$ for a SSSP.
Therefore, it takes $O(|V|^2 log|V| + |V||E|)$ for APSP.
It can't used with a negative weight but it can be fixed by Johnson's algorithm.

Introduction

Bellman-Ford algorithm

```
function BellmanFord(list vertices, list edges, vertex source) is
    ::distance[], predecessor[]

    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v] := inf
        predecessor[v] := null

    distance[source] := 0

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1 do
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u

    return distance[], predecessor[]
```

Figure 11. Bellman-ford algorithm

Bellman-Ford algorithm can solve with negative weighted edges.

It takes $O(|V||E|)$ for a SSSP.

Therefore, it takes $O(|V|^2|E|)$ for APSP.

Introduction

Dijkstra’s algorithm versus Bellman-ford algorithm

Algorithm	**Complexity(APSP)**	**With negative edges**
Dijkstra’s algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	No
Bellman-ford algorithm	$O(\|V\|^2\|E\|)$	Yes

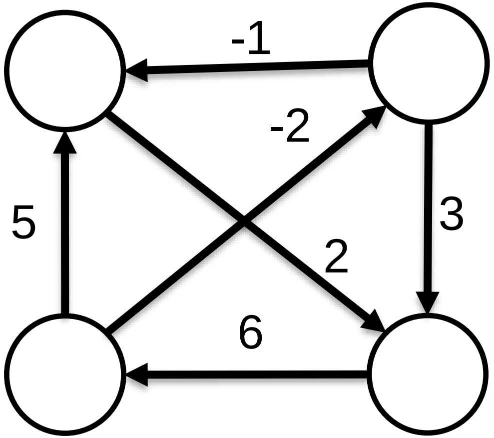
Introduction

Johnson's algorithm

Algorithm	Complexity(APSP)	With negative edges
Dijkstra's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	No
Bellman-ford algorithm	$O(\|V\|^2\|E\|)$	Yes
Johnson's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	Yes

Introduction

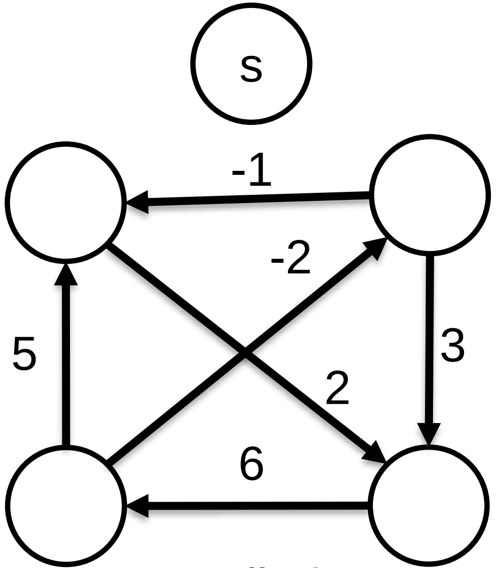
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u, v) + h(u) - h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

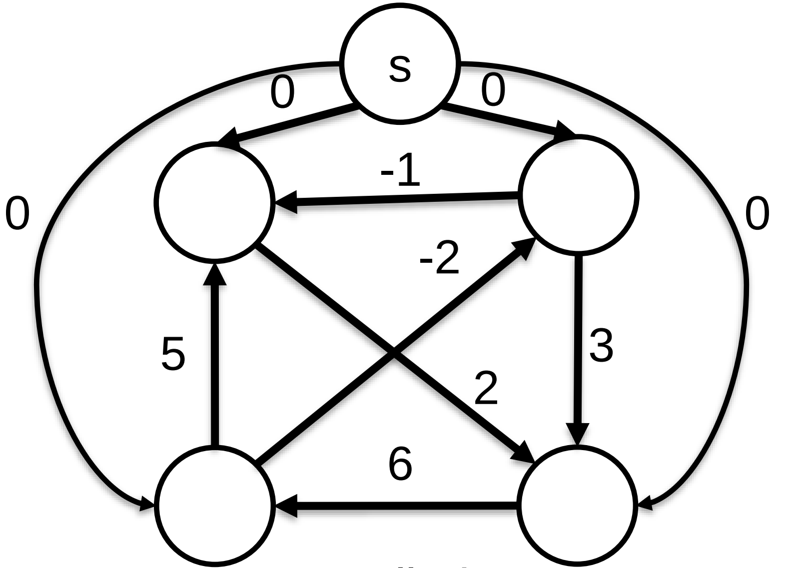
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u, v) + h(u) - h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

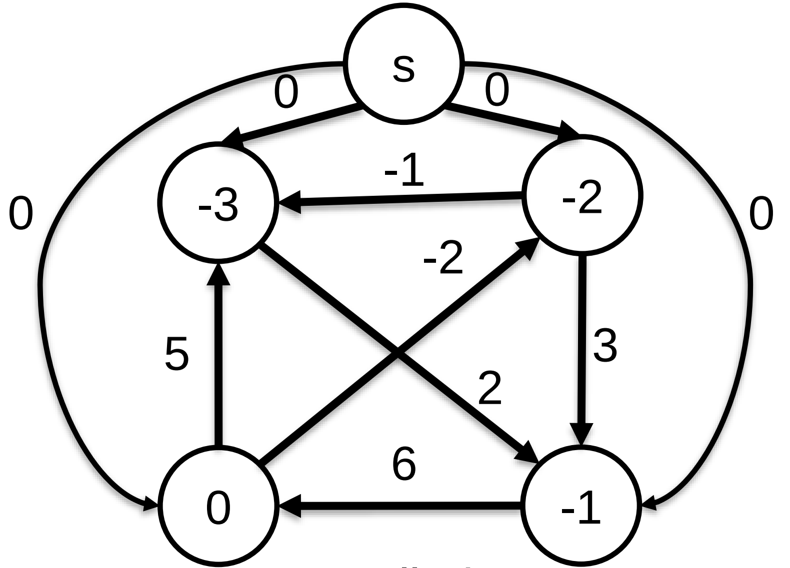
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v)+h(u)-h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

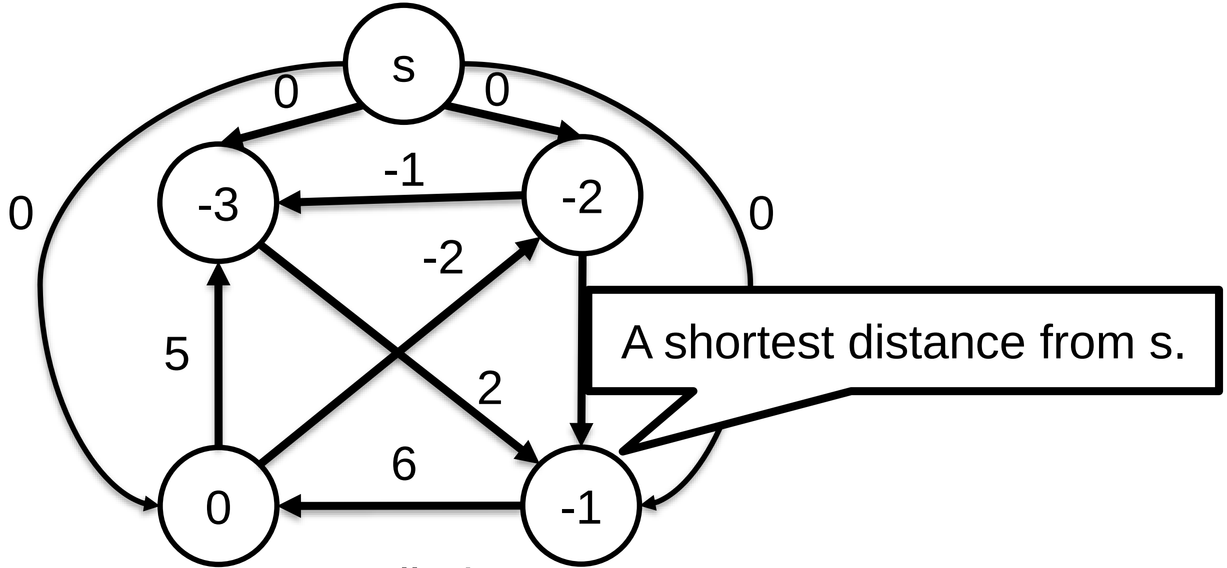
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v)+h(u)-h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

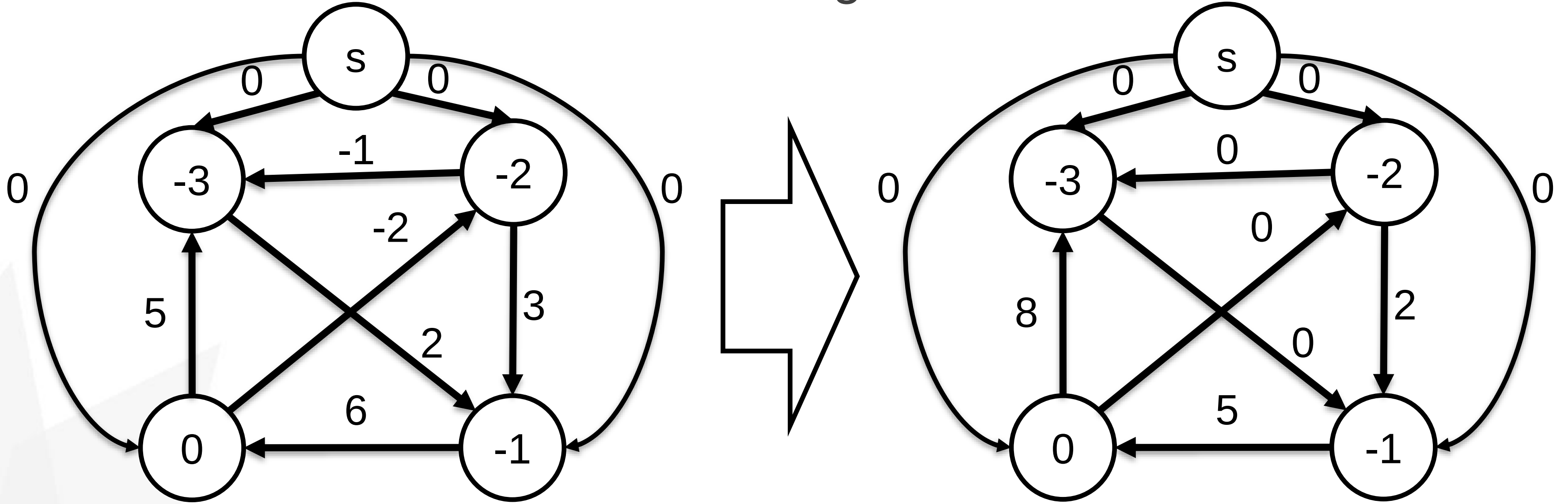
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u, v) + h(u) - h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

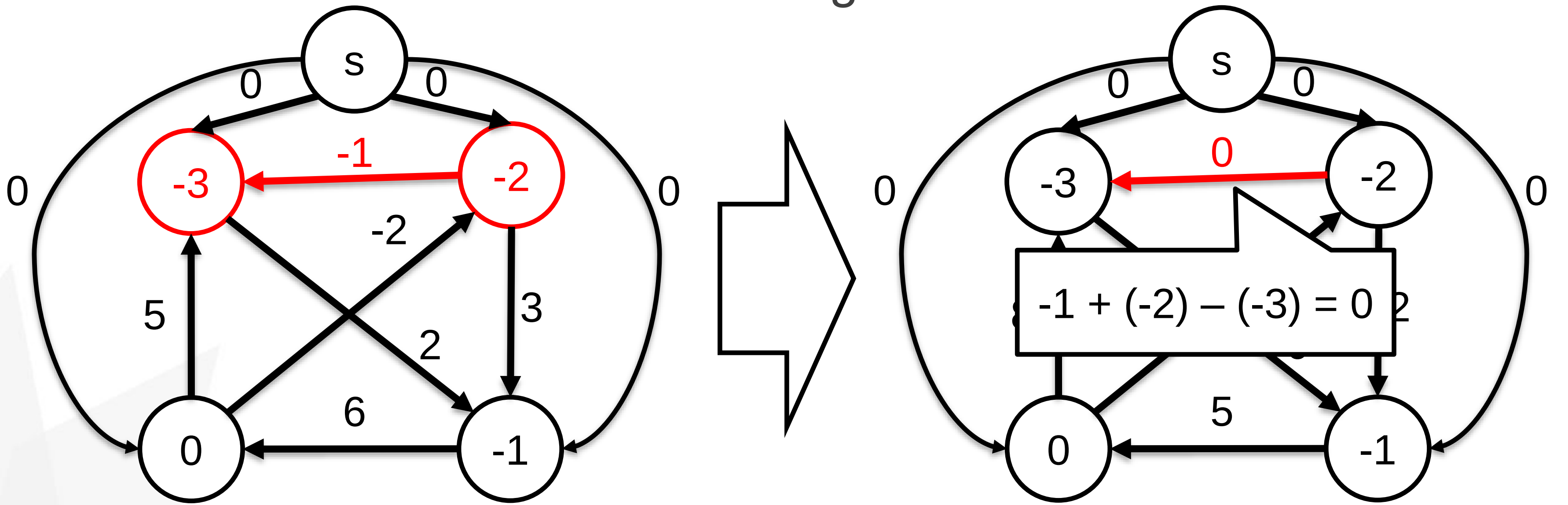
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v)+h(u)-h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

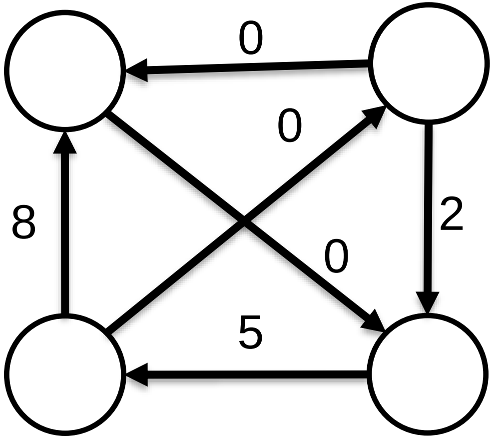
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v) + h(u) - h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

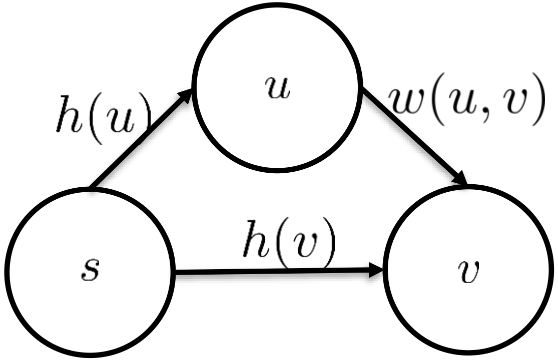
Johnson's algorithm



1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v) + h(u) - h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

Johnson's algorithm

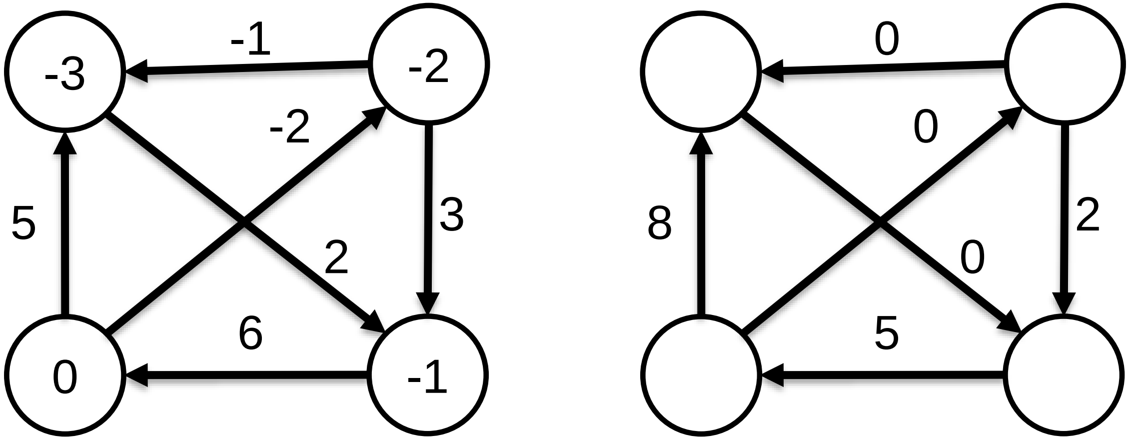


3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v)+h(u)-h(v)$

Since, it proceed Bellman-Ford, $w(u,v)+h(u)\geq h(v)$.
Therefore, it has no negative edges.

Introduction

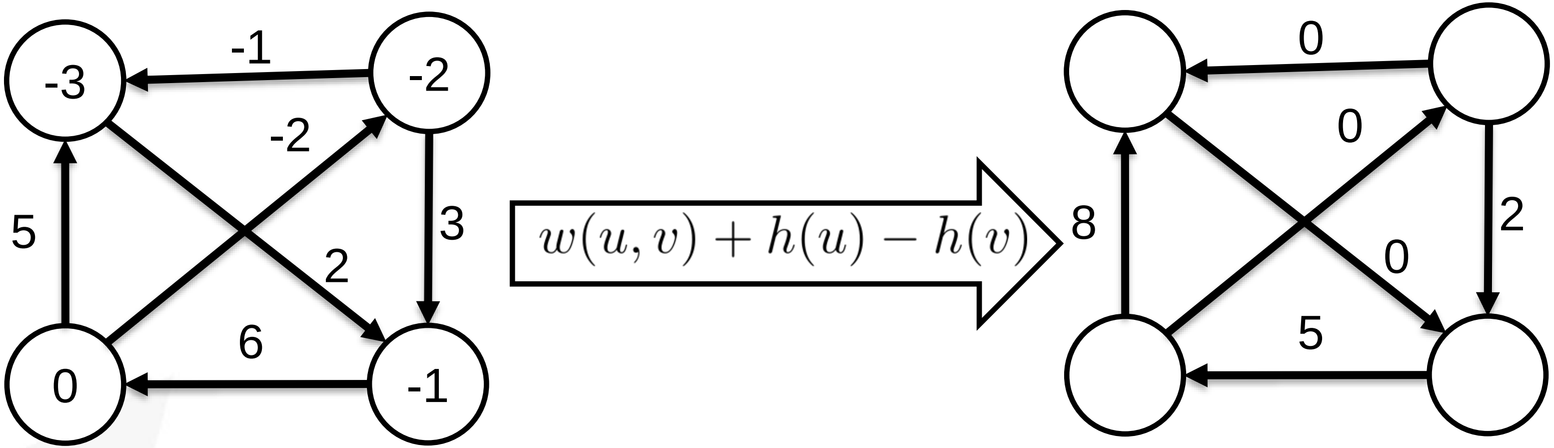
Johnson's algorithm



The cost of path could be changed, but it reserves shortest paths.
Example of shortest path in both graphs are above.
Proof will be follows.

Introduction

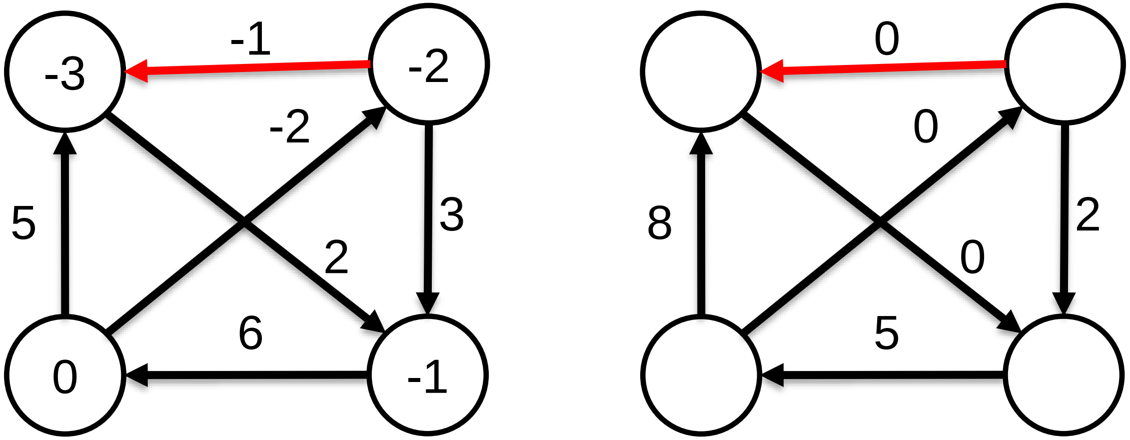
Johnson's algorithm



The cost of path could be changed, but it reserves shortest paths.
Example of shortest path in both graphs are above.
Proof will be follows.

Introduction

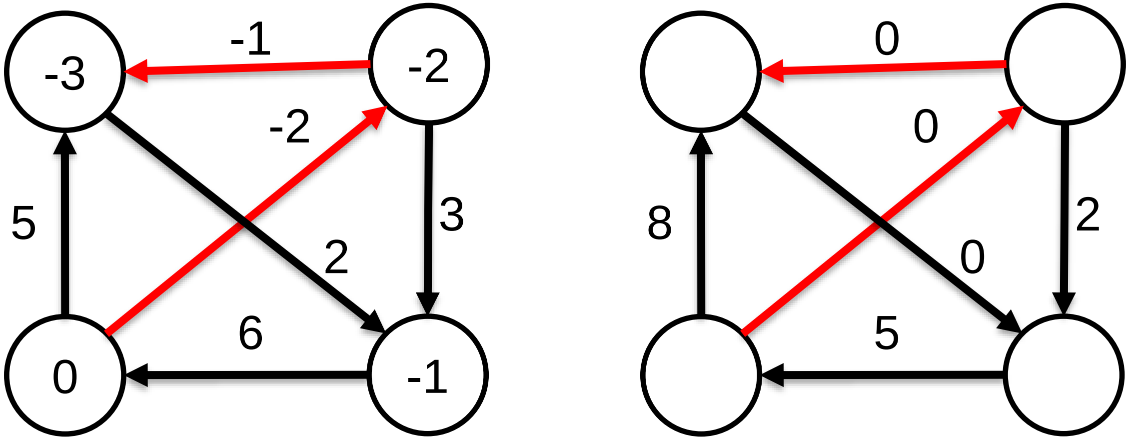
Johnson's algorithm



The cost of path could be changed, but it reserves shortest paths.
Example of shortest path in both graphs are above.
Proof will be follows.

Introduction

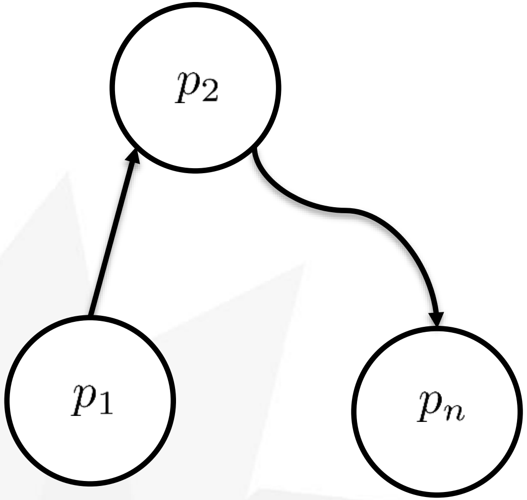
Johnson’s algorithm



The cost of path could be changed, but it reserves shortest paths.
Example of shortest path in both graphs are above.
Proof will be follows.

Introduction

Johnson's algorithm



Correctness proof is as follow.

Let's think about path $P = (p_1, p_2, \cdots, p_n)$.

The path cost between is p_1 and p_n is

$$
\begin{aligned}
&(w(p_1, p_2) + h(p_1) - h(p_2)) + (w(p_2, p_3) + h(p_2) - h(p_3)) \\
&+ \cdots + (w(p_{n-1}, p_n) + h(p_{n-1}) - h(p_n)) \\
&= w(p_1, p_2) + w(p_2, p_3) + \cdots + w(p_{n-1}, p_n) + (h(p_1) - h(p_2)) \\
&+ (h(p_2) - h(p_3)) + \cdots + (h(p_{n-1}) - h(p_n)) \\
&= w(p_1, p_2) + \cdots + w(p_{n-1}, p_n) + h(p_1) - h(p_n) \\
&= w(p_1, p_2) + \cdots + w(p_{n-1}, p_n) + h(p_1) - h(p_n)
\end{aligned}
$$

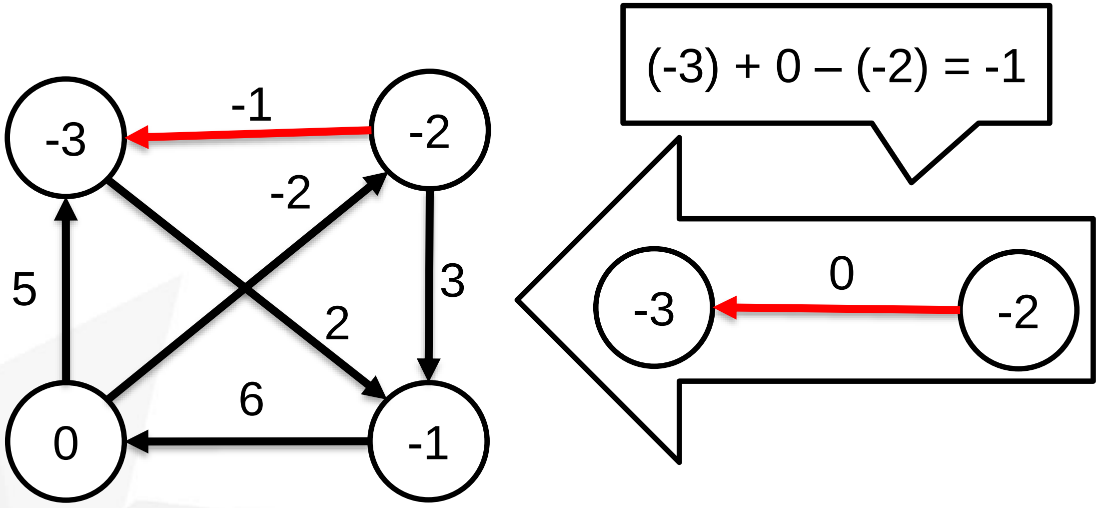
It implies the fact that path cost will increase by $h(p_1) - h(p_n)$.

Which only depend on source and destination.

Therefore, it still reserves shortest path.

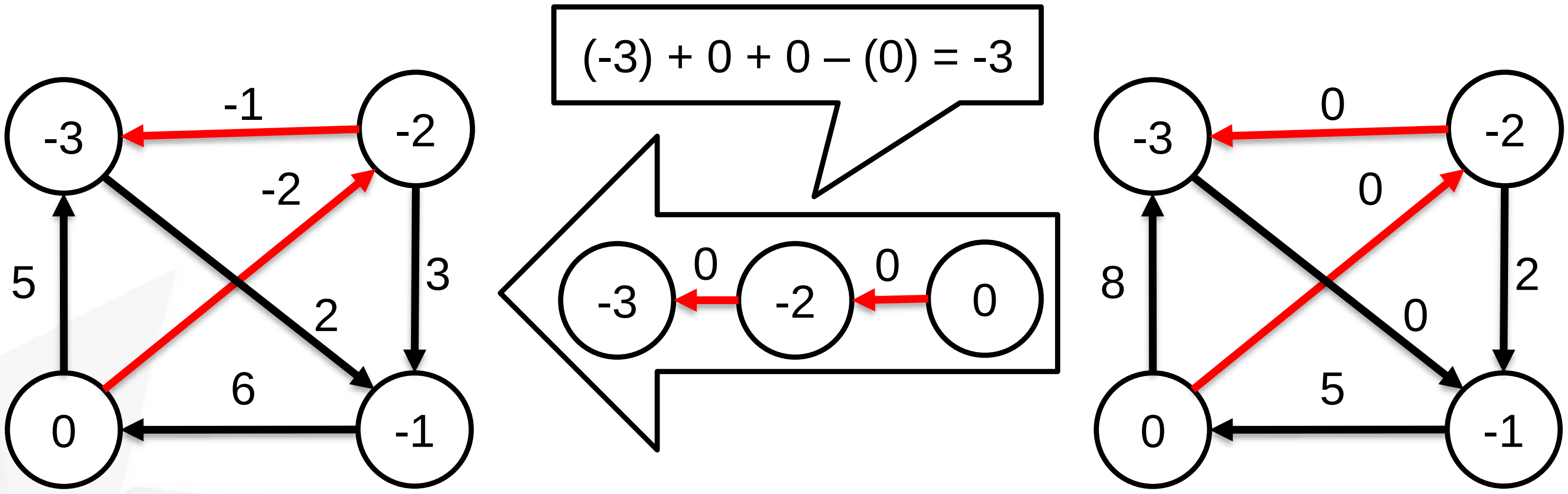
Introduction

Johnson's algorithm



Introduction

Johnson's algorithm



Introduction

Johnson’s algorithm

Algorithm	Complexity(SSSP)	With negative edges
Dijkstra’s algorithm	$O(\|V\|log\|V\| + \|E\|)$	No
Bellman-ford algorithm	$O(\|V\|\|E\|)$	Yes

Introduction

Johnson's algorithm

$O(1)$

$O(|V|)$

$O(|V|(|V| + |E|))$

$O(|E|)$

$O(|V|^2 log|V| + |V||E|)$

1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u, v) + h(u) - h(v)$
5. Do Dijkstra's algorithm to solve APSP

Introduction

Johnson's algorithm

$O(1)$

SSSP

$O(|V|)$

$O(|V|(|V|+|E|))$

1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v)+h(u)-h(v)$
5. Do Dijkstra's algorithm to solve APSP

$O(|E|)$

APSP

$O(|V|^2 log|V| + |V||E|)$

Introduction

Johnson's algorithm

$O(1)$

$O(|V|)$

$O(|V|(|V|+|E|))$

1. Make a new source called s.
2. Connect s with all other vertices with cost of 0.
3. Do Bellman-ford algorithm from s and save it to h .
4. Update weights as follow $w(u,v)+h(u)-h(v)$
5. Do Dijkstra's algorithm to solve APSP

$O(|E|)$

$O(|V|^2 log|V| + |V||E|)$

Introduction

Johnson's algorithm

Algorithm	**Complexity(APSP)**	**With negative edges**
Dijkstra's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	No
Bellman-ford algorithm	$O(\|V\|^2\|E\|)$	Yes
Johnson's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	Yes

Sparse graph can consider $E = O(V)$.

Introduction

Johnson's algorithm

Algorithm	Complexity(APSP)	With negative edges
Dijkstra's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	No
Bellman-ford algorithm	$O(\|V\|^2\|E\|)$	Yes
Johnson's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	Yes

Sparse graph can consider $E = O(V)$.

Algorithm	Complexity(APSP)	With negative edges
Dijkstra's algorithm	$O(\|V\|^2 log\|V\|)$	No
Bellman-ford algorithm	$O(\|V\|^3)$	Yes
Johnson's algorithm	$O(\|V\|^2 log\|V\|)$	Yes

Introduction

Johnson's algorithm

Algorithm	Complexity(APSP)	With negative edges
Dijkstra's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	No
Bellman-ford algorithm	$O(\|V\|^2\|E\|)$	Yes
Johnson's algorithm	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	Yes

Sparse graph can consider $E = O(V)$.

Algorithm	Complexity(APSP)	With negative edges
Dijkstra's algorithm		No
Bellman-ford algorithm	$O(\|V\|^3)$	Yes
Johnson's algorithm	$O(\|V\|^2 log\|V\|)$	Yes

Algorithm used for sparse graphs.

Introduction

Johnson's algorithm

As a result, Johnson's algorithm complexity is Dijkstra + Bellman-ford.
It takes $O(|V|^2 log|V| + |V||E|)$.

In case $E = O(V)$ (Sparse), It takes $O(|V|^2 log|V|)$.
It may faster for such a case.

However, Johnson's algorithm doesn't have a nice parallelism.
It underutilizes the performance of modern architecture.

Introduction

Floyd warshall algorithm

Algorithm 1 Floyd-Warshall algorithm for APSP

function FloydWarshall($G=(V,E)$):
  Let $n \leftarrow \dim(V)$
  Let $\mathrm{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherwise} \end{cases}$
  for $k = \{1, 2\ldots,n\}$ **do**:
    for $i = \{1, 2\ldots,n\}$ **do**:
      for $j = \{1, 2\ldots,n\}$ **do**:
        $\mathrm{Dist}[i,j] = \min\{\mathrm{Dist}[i,j],\ \mathrm{Dist}[i,k] + \mathrm{Dist}[k,j]\}$
  Return Dist

Figure 12. Floyd-warshall algorithm

Floyd-Warshall algorithm solves APSP by cooperating each other.
It takes $O(|V|^3)$, but shows more potential for parallelism.

Introduction

Floyd warshall algorithm

Algorithm 1 FLOYD-WARSHALL algorithm for APSP

```
function FLOYDWARSHALL(G = (V, E)):
    Let n ← dim(V)
    Let Dist[i,j] = { w_{i,j}  if (i,j) ∈ E
                    { ∞        otherwise
    for k = {1, 2…, n} do:
        for i = {1, 2…, n} do:
            for j = {1, 2…, n} do:
                Dist[i,j] = min{Dist[i,j], Dist[i,k] + Dist[k,j]}
    Return Dist
```

$$\text{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Figure 12. Floyd-warshall algorithm

Introduction

Floyd warshall algorithm

Algorithm 1 FLOYD-WARSHALL algorithm for APSP

function FLOYDWARSHALL($G=(V,E)$):

Let $n \leftarrow \dim(V)$

Let $\mathrm{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherwise} \end{cases}$

Initialization

for $k = \{1, 2\ldots,n\}$ **do**:

for $i = \{1, 2\ldots,n\}$ **do**:

for $j = \{1, 2\ldots,n\}$ **do**:

$\mathrm{Dist}[i,j] = \min\{\mathrm{Dist}[i,j],\ \mathrm{Dist}[i,k]+\mathrm{Dist}[k,j]\}$

Return Dist

Figure 12. Floyd-warshall algorithm

Introduction

Floyd warshall algorithm

Algorithm 1 FLOYD-WARSHALL algorithm for APSP

function FLOYDWARSHALL($G=(V,E)$):

  Let $n \leftarrow \dim(V)$

  Let $\mathrm{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherw} \end{cases}$

  $\boxed{\textbf{for } k = \{1, 2\ldots,n\} \textbf{ do:}}$ For each intermediaries

    for $i = \{1, 2\ldots,n\}$ **do**:

      for $j = \{1, 2\ldots,n\}$ **do**:

        $\mathrm{Dist}[i,j] = \min\{\mathrm{Dist}[i,j],\ \mathrm{Dist}[i,k] + \mathrm{Dist}[k,j]\}$

  Return Dist

Figure 12. Floyd-warshall algorithm

Introduction

Floyd warshall algorithm

Algorithm 1 FLOYD-WARSHALL algorithm for APSP

function FLOYDWARSHALL($G = (V, E)$):

  Let $n \leftarrow \dim(V)$

  Let $\mathrm{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherwise} \end{cases}$

  for $k = \{1, 2\ldots, n\}$ **do**:

    for $i = \{1, 2\ldots, n\}$ **do**:

      for $j = \{1, 2\ldots, n\}$ **do**:

        $\mathrm{Dist}[i,j] = \min\{\mathrm{Dist}[i,j],\ \mathrm{Dist}[i,k] + \mathrm{Dist}[k,j]\}$

  Return Dist

For all pairs

Figure 12. Floyd-warshall algorithm

Introduction

Floyd warshall algorithm

Algorithm 1 FLOYD-WARSHALL algorithm for APSP

function FLOYDWARSHALL($G=(V,E)$):

  Let $n \leftarrow \dim(V)$

  Let $\mathrm{Dist}[i,j] = \begin{cases} w_{i,j} & \text{if}(i,j) \in E \\ \infty & \text{otherwise} \end{cases}$

  for $k = \{1, 2 \ldots, n\}$ **do**:

    for $i = \{1, 2 \ldots, n\}$ **do**:

      for $j = \{1, 2 \ldots, n\}$ **do**:

        $\mathrm{Dist}[i,j] = \min\{\mathrm{Dist}[i,j],\ \mathrm{Dist}[i,k] + \mathrm{Dist}[k,j]\}$

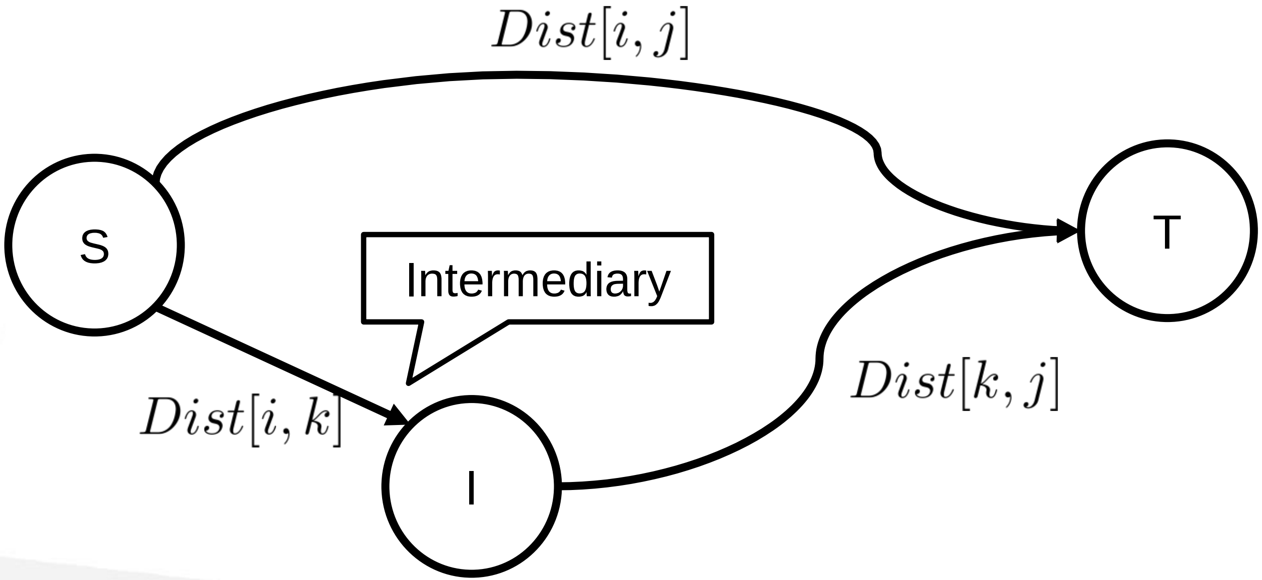
Update distances with intermediaries

  Return Dist

Figure 12. Floyd-warshall algorithm

Introduction

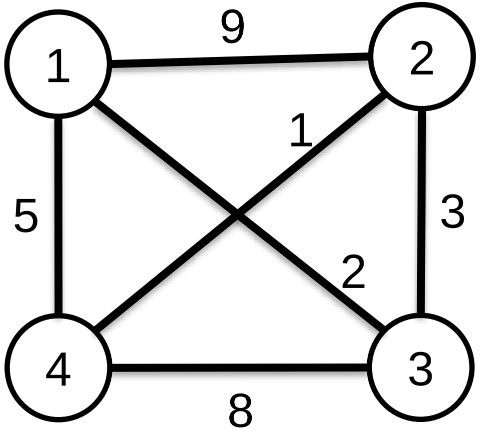
Floyd warshall algorithm



$$Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])$$

Introduction

Floyd warshall algorithm



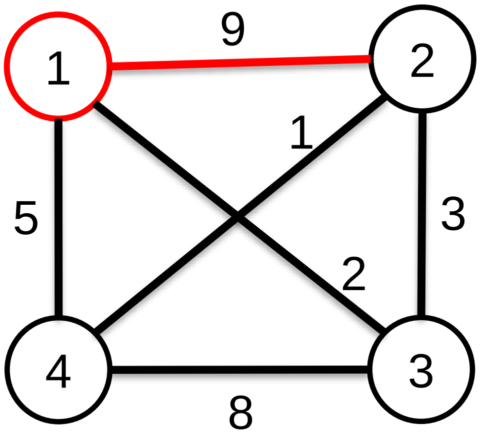
	1	2	3	4
1	X	9	2	5
2	9	X	3	1
3	2	3	X	8
4	5	1	8	X

Figure 13. Graph and distance matrix

$$k = 1$$
$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



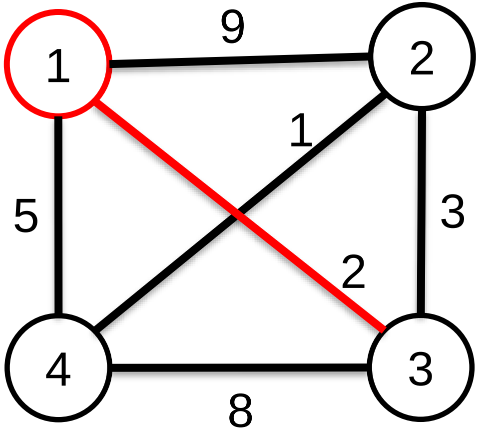
	1	2	3	4
1	X	9	2	5
2	9	18	3	1
3	2	3	X	8
4	5	1	8	X

Figure 13. Graph and distance matrix

$$k = 1$$
$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



	1	**2**	**3**	**4**
1	X	9	2	5
2	9	18	3	1
3	2	3	4	8
4	5	1	8	X

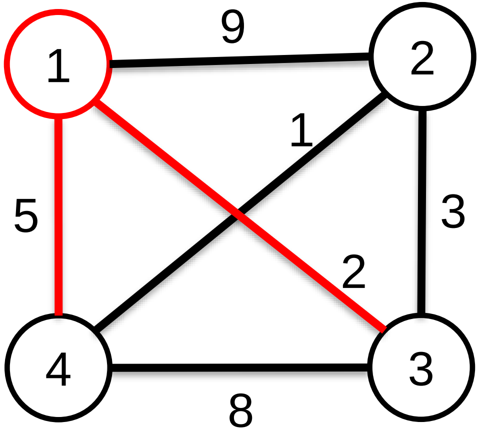
Figure 13. Graph and distance matrix

$$k = 1$$

$$Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])$$

Introduction

Floyd warshall algorithm



	1	2	3	4
1	X	9	2	5
2	9	18	3	1
3	2	3	4	7
4	5	1	7	X

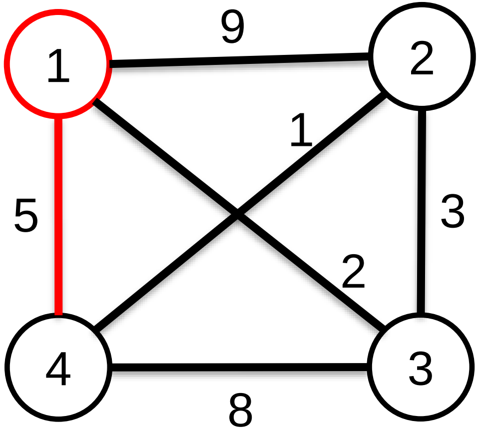
Figure 13. Graph and distance matrix

$$k = 1$$

$$Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])$$

Introduction

Floyd warshall algorithm



	1	2	3	4
1	X	9	2	5
2	9	18	3	1
3	2	3	4	7
4	5	1	7	10

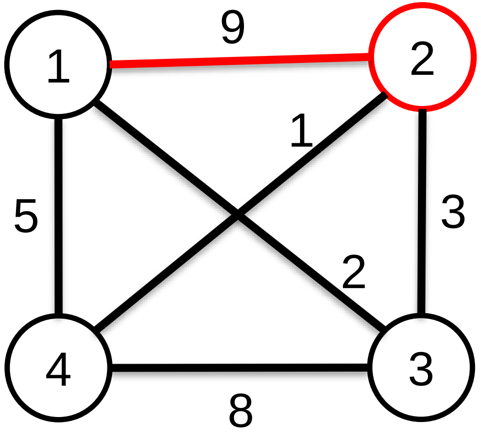
Figure 13. Graph and distance matrix

$$k = 1$$

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



	1	2	3	4
1	18	9	2	5
2	9	18	3	1
3	2	3	4	7
4	5	1	7	10

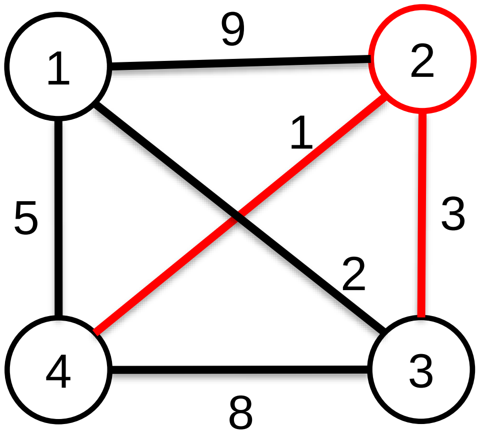
Figure 13. Graph and distance matrix

$$k = 2$$

$$Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])$$

Introduction

Floyd warshall algorithm



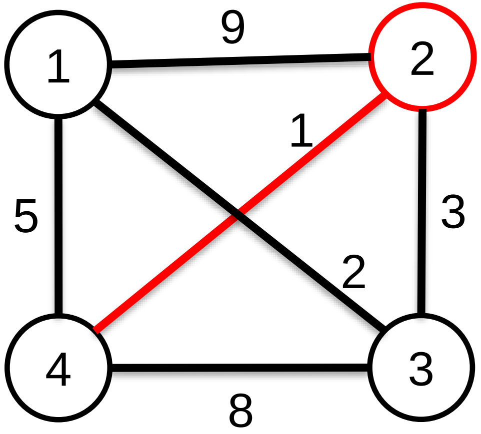
	1	2	3	4
1	18	9	2	5
2	9	18	3	1
3	2	3	4	4
4	5	1	4	10

Figure 13. Graph and distance matrix

$$k = 2$$
$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



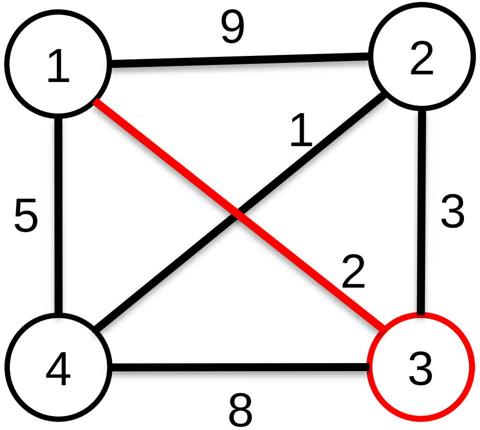
	1	2	3	4
1	18	9	2	5
2	9	18	3	1
3	2	3	4	4
4	5	1	4	2

Figure 13. Graph and distance matrix

$$k = 2$$
$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



	1	2	3	4
1	4	9	2	5
2	9	18	3	1
3	2	3	4	4
4	5	1	4	2

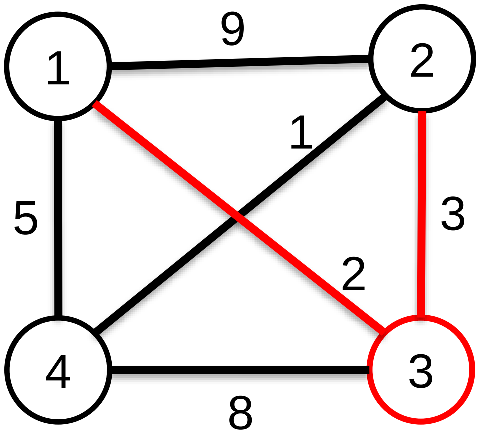
Figure 13. Graph and distance matrix

$$k = 3$$

$$Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])$$

Introduction

Floyd warshall algorithm



	1	2	3	4
1	4	5	2	5
2	5	18	3	1
3	2	3	4	4
4	5	1	4	2

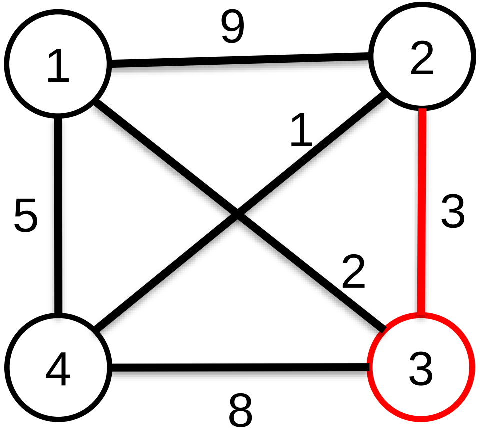
Figure 13. Graph and distance matrix

$$k = 3$$

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



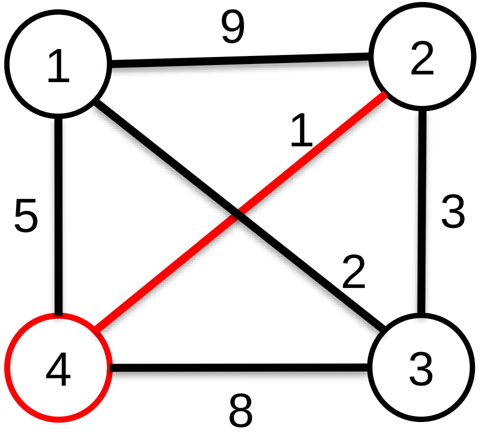
	1	**2**	**3**	**4**
1	4	5	2	5
2	5	6	3	1
3	2	3	4	4
4	5	1	4	2

Figure 13. Graph and distance matrix

$$k = 3$$
$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Introduction

Floyd warshall algorithm



	1	2	3	4
1	4	5	2	5
2	5	2	3	1
3	2	3	4	4
4	5	1	4	2

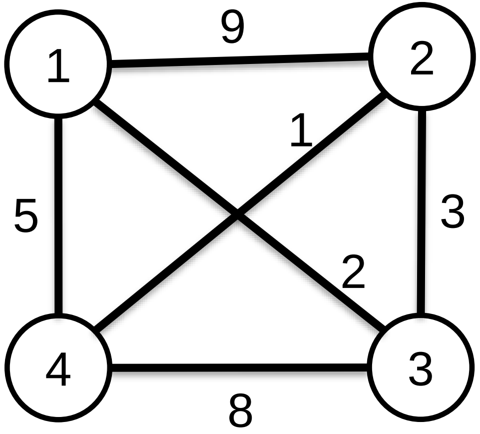
Figure 13. Graph and distance matrix

$$k = 4$$

$$Dist[i, j] = min(Dist[i, j], Dist[i, k] + Dist[k, j])$$

Introduction

Floyd warshall algorithm



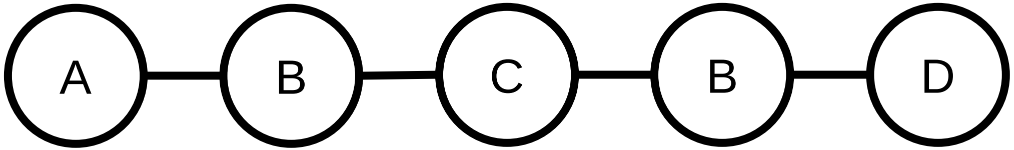
	1	**2**	**3**	**4**
1	4	5	2	5
2	5	2	3	1
3	2	3	4	4
4	5	1	4	2

Figure 13. Graph and distance matrix

Now it's done.
However… how does it work?

Introduction

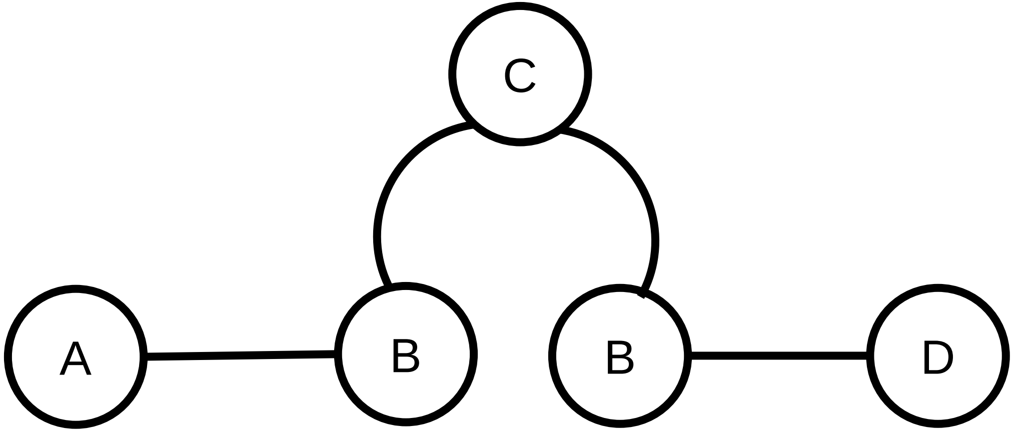
Floyd warshall algorithm



There is a shortest path with out cycle.
Notice that there is no negative cycle in APSP.

Introduction

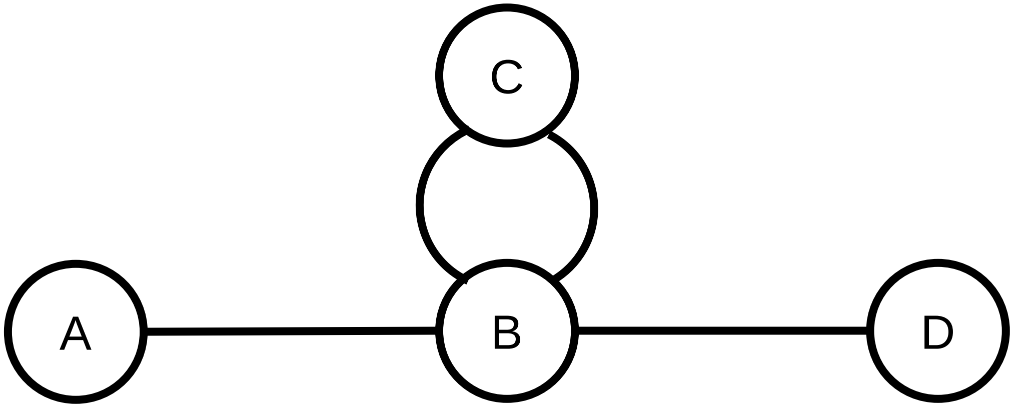
Floyd warshall algorithm



There is a shortest path with out cycle.
Notice that there is no negative cycle in APSP.

Introduction

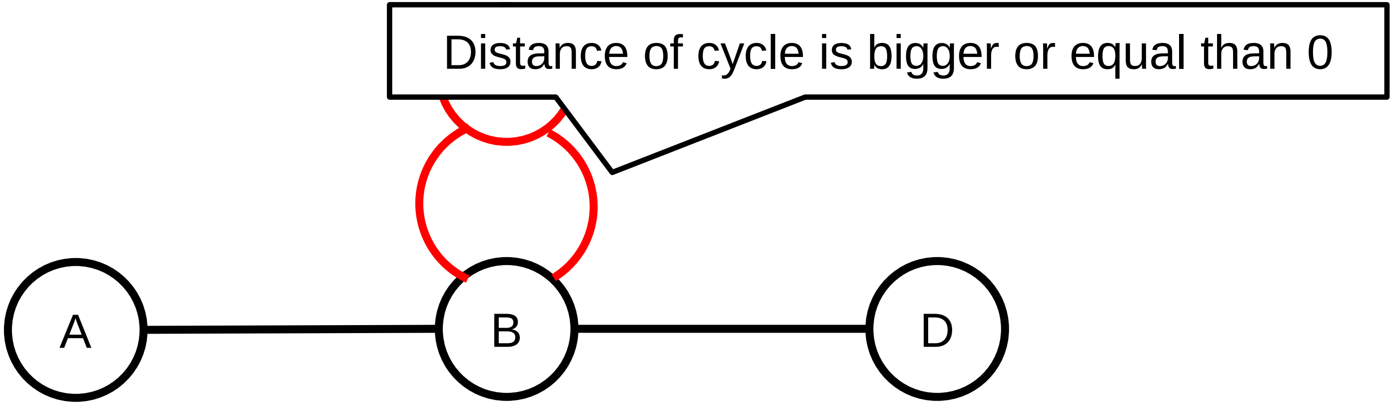
Floyd warshall algorithm



There is a shortest path with out cycle.
Notice that there is no negative cycle in APSP.

Introduction

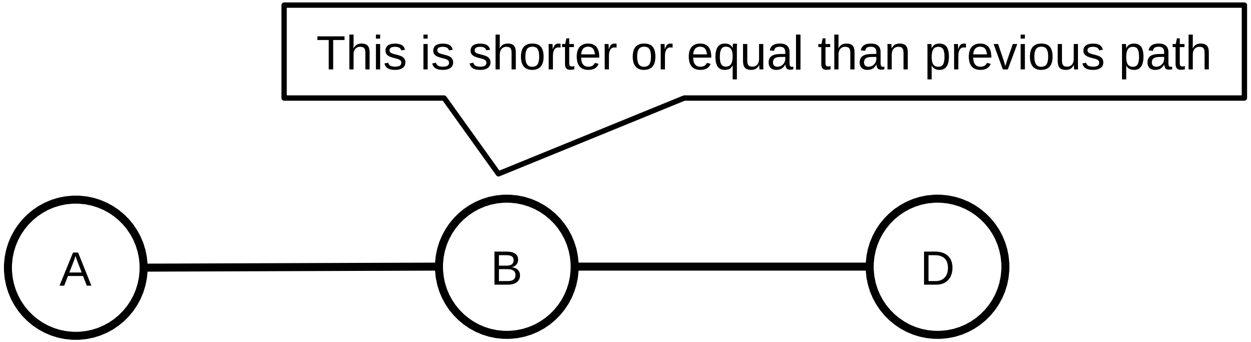
Floyd warshall algorithm



There is a shortest path with out cycle.
Notice that there is no negative cycle in APSP.

Introduction

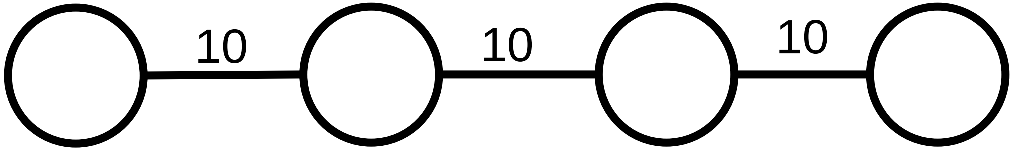
Floyd warshall algorithm



There is a shortest path with out cycle.
Notice that there is no negative cycle in APSP.

Introduction

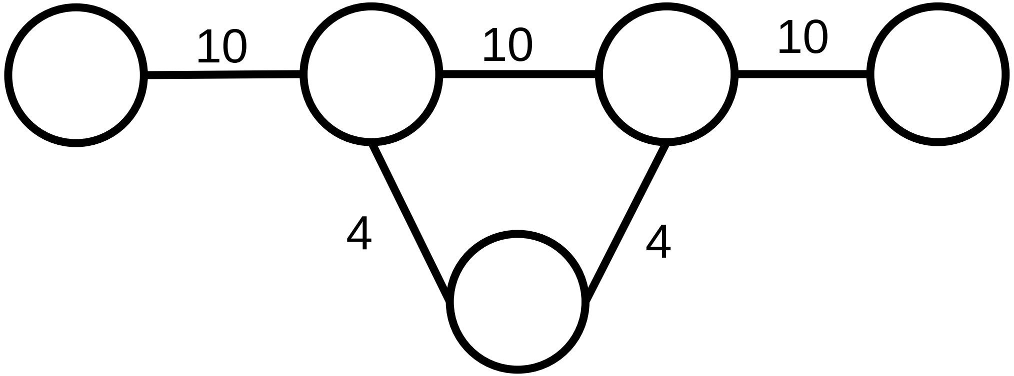
Floyd warshall algorithm



Subpaths of the shortest path are the shortest.

Introduction

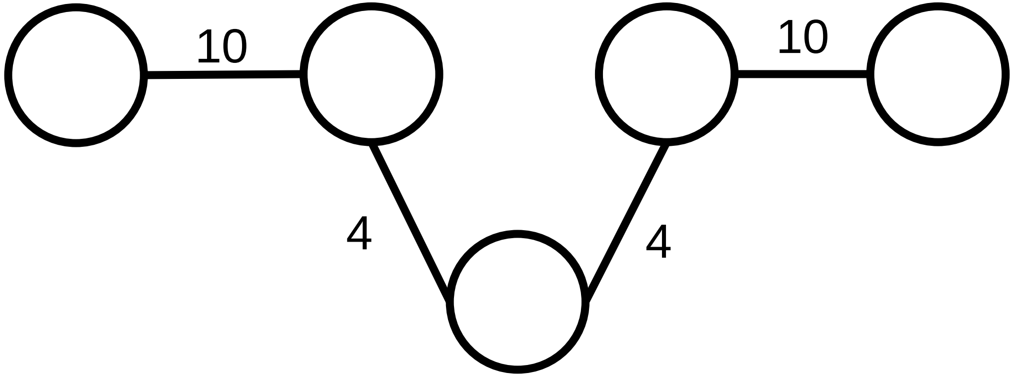
Floyd warshall algorithm



Subpaths of the shortest path are the shortest.

Introduction

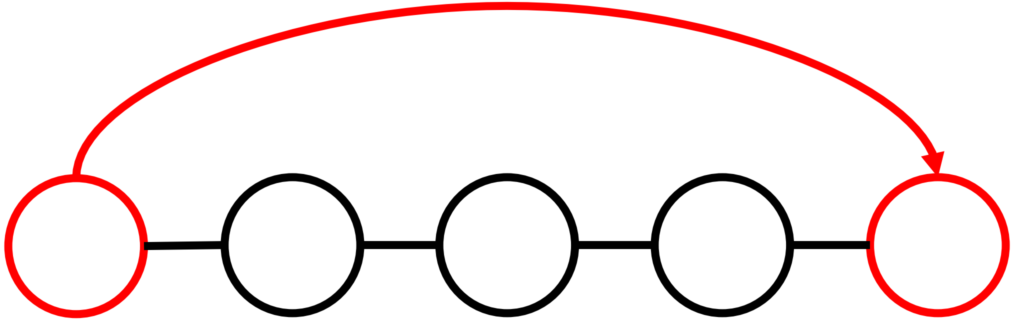
Floyd warshall algorithm



Subpaths of the shortest path are the shortest.

Introduction

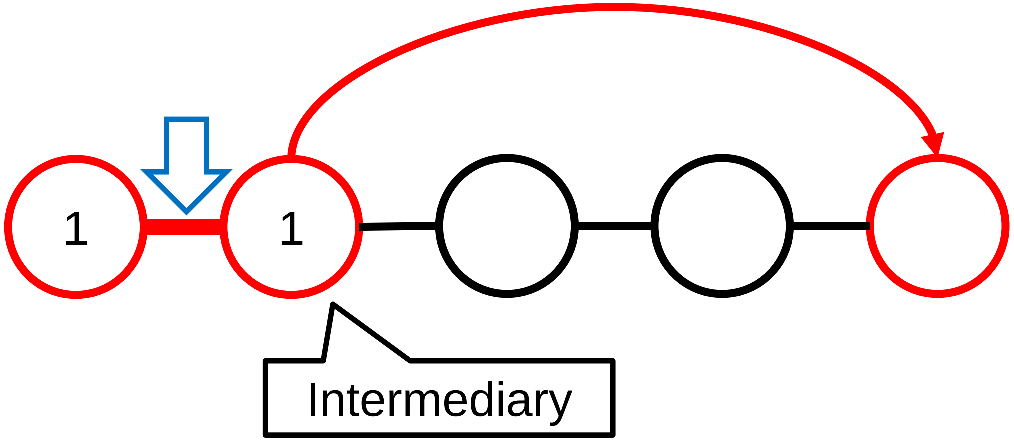
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

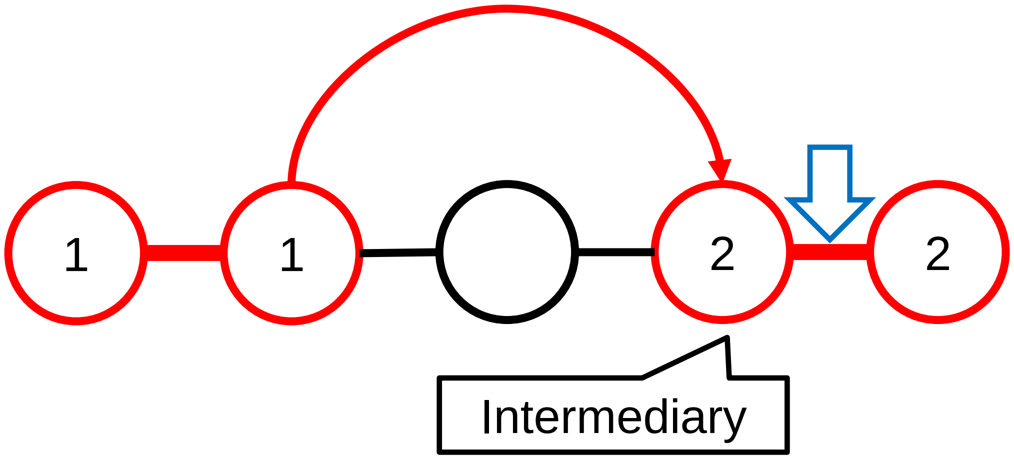
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

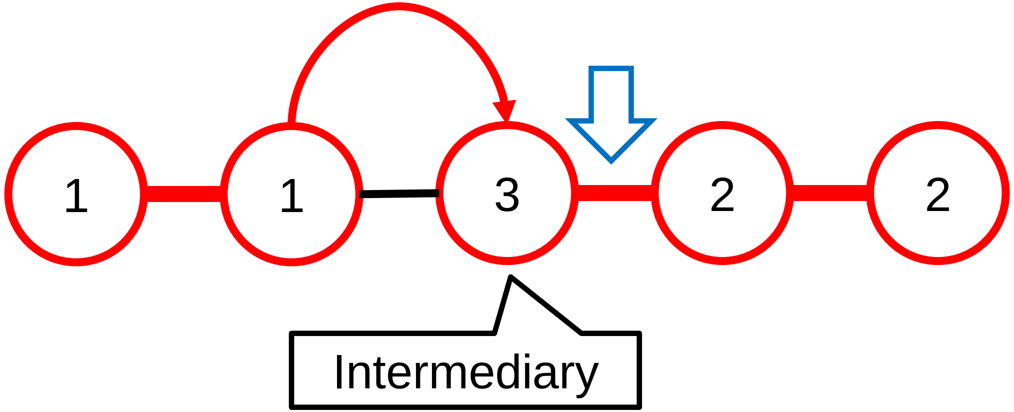
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

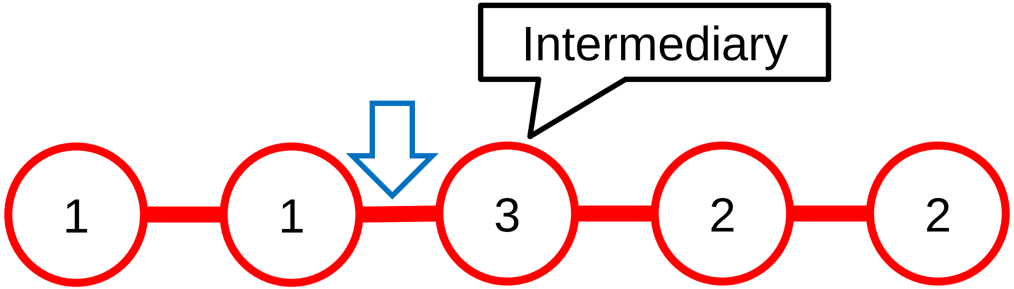
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

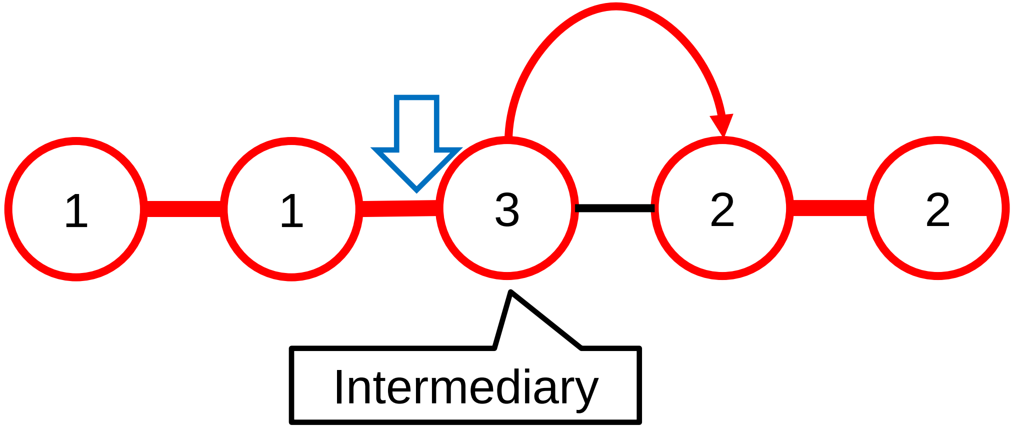
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

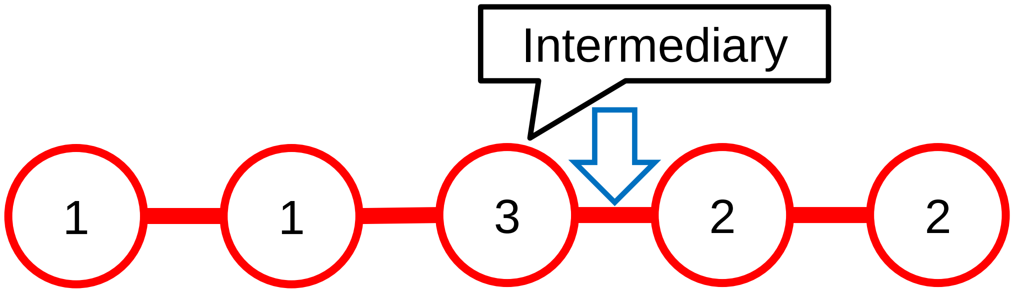
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

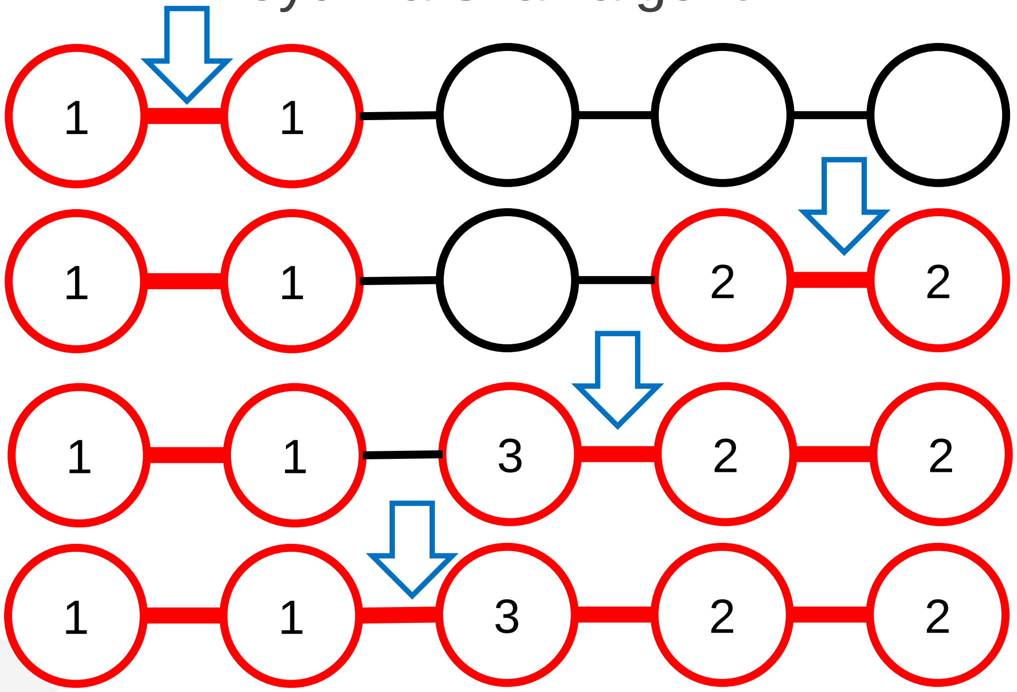
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

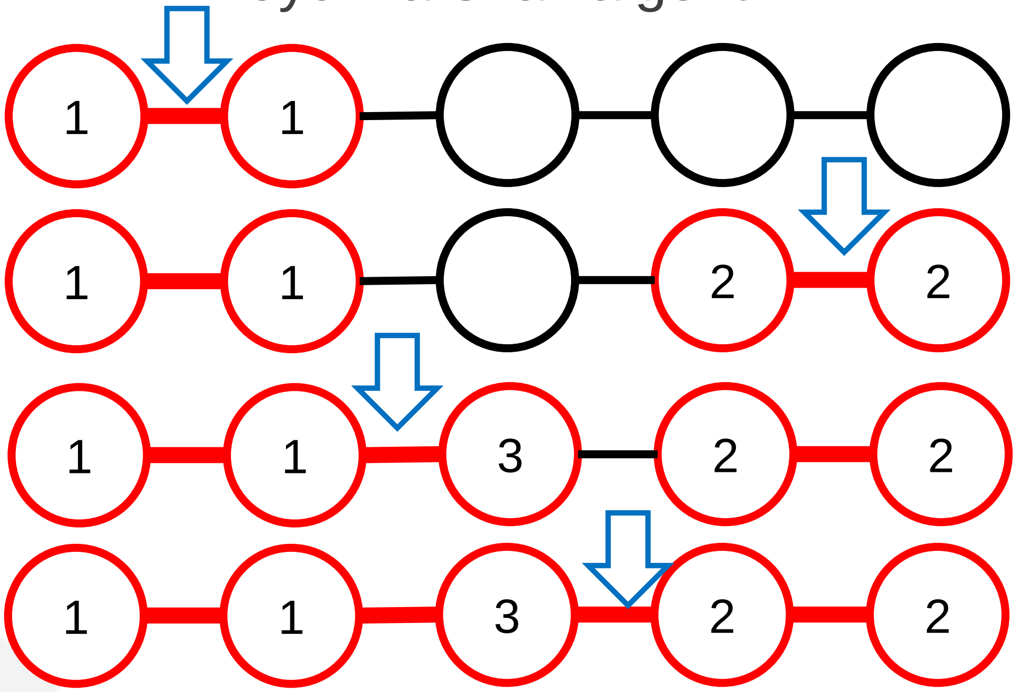
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

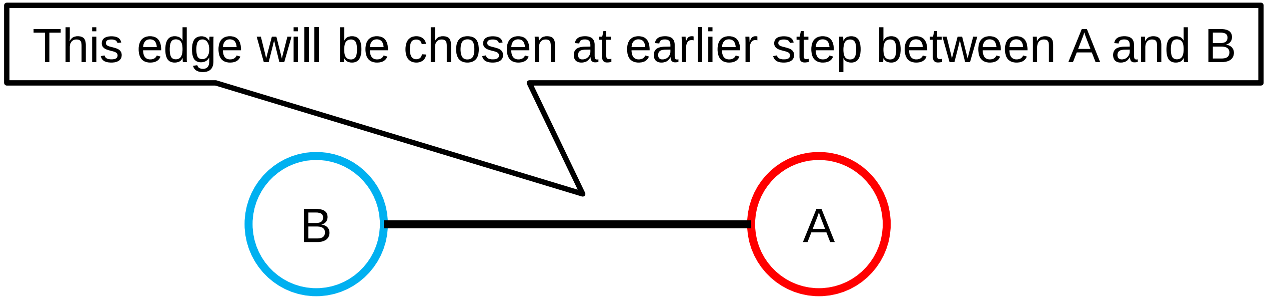
Floyd warshall algorithm



Notice that a sequence to chose edges of the shortest path exists.
This sequence can be done in an arbitrary order.

Introduction

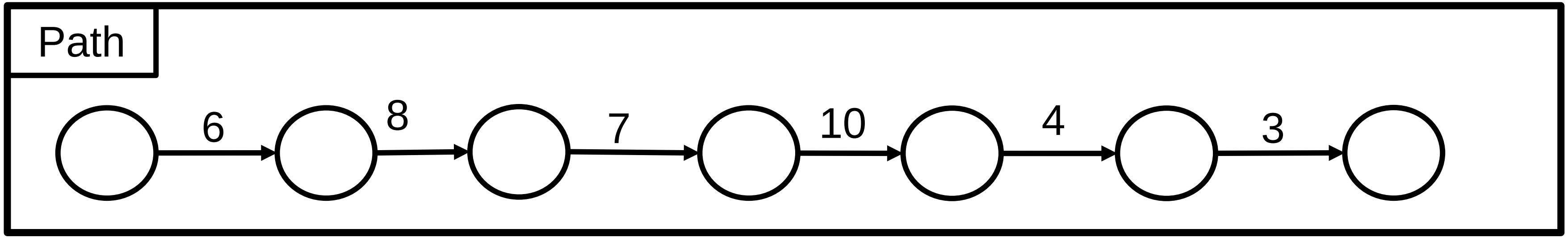
Floyd warshall algorithm

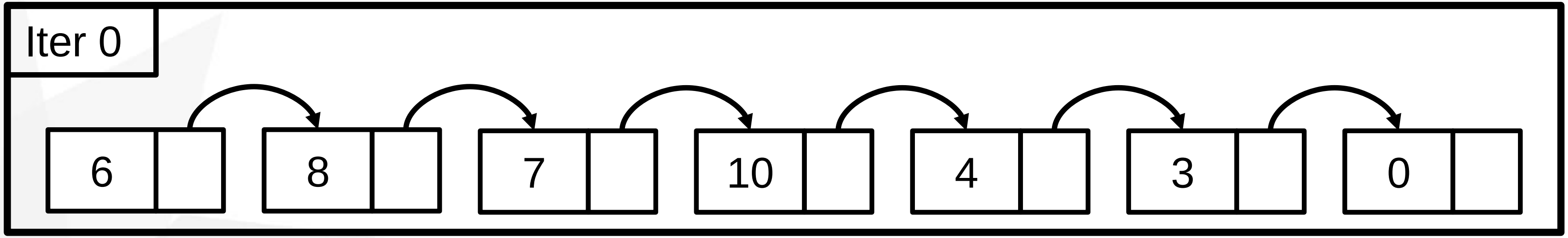


Notice that there is a shortest path with out cycle for all-pairs of vertices.
Therefore, A and B is always different.
Arbitrary order of vertices can be chosen to produce shortest path.
Floyd-warshall algorithm uses an order of vetices for APSP.

Background

Min-Plus Matrix Multiplication

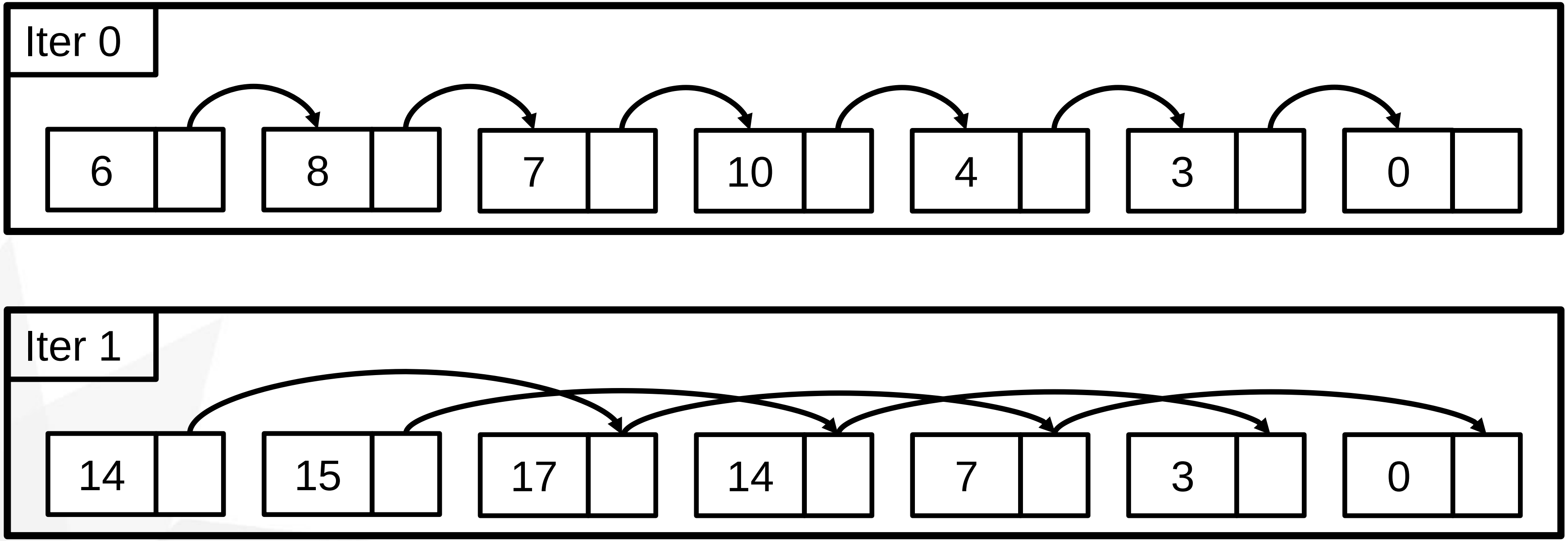




Path doubling algorithm(or Path jumping) is an algorithm that can coumpte a distance of edge in a logarithmic time.

Background

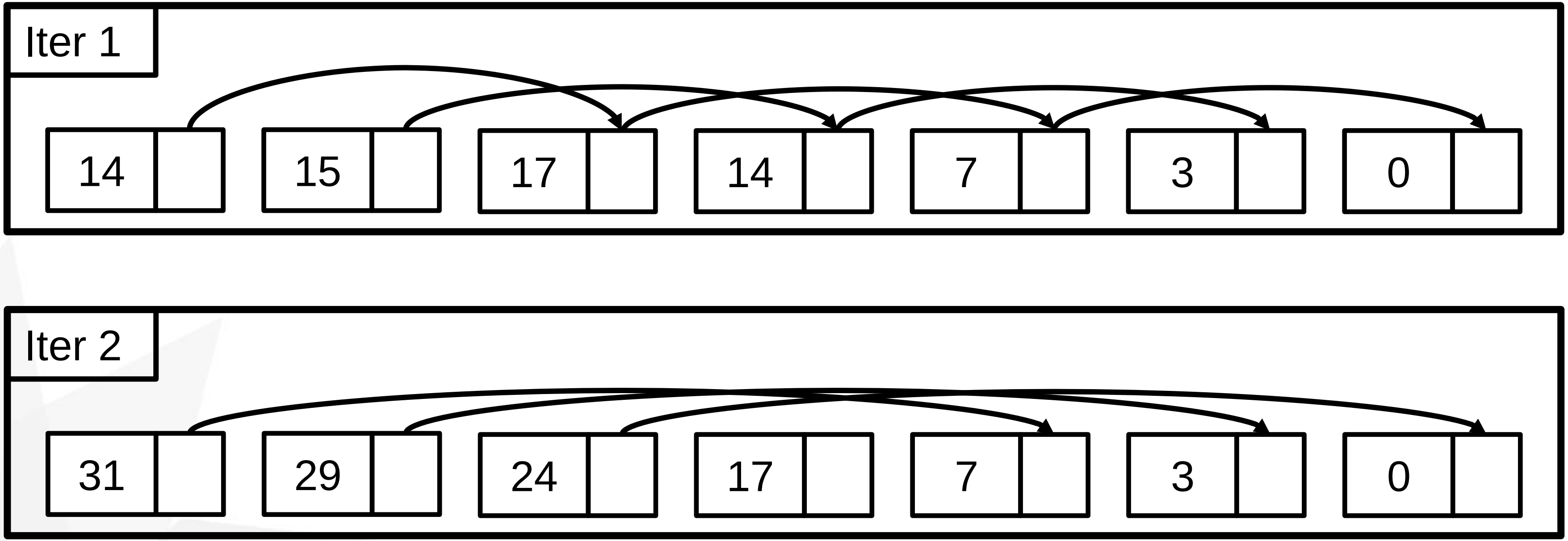
Min-Plus Matrix Multiplication



Path doubling algorithm(or Path jumping) is an algorithm that can coumpte a distance of edge in a logarithmic time.

Background

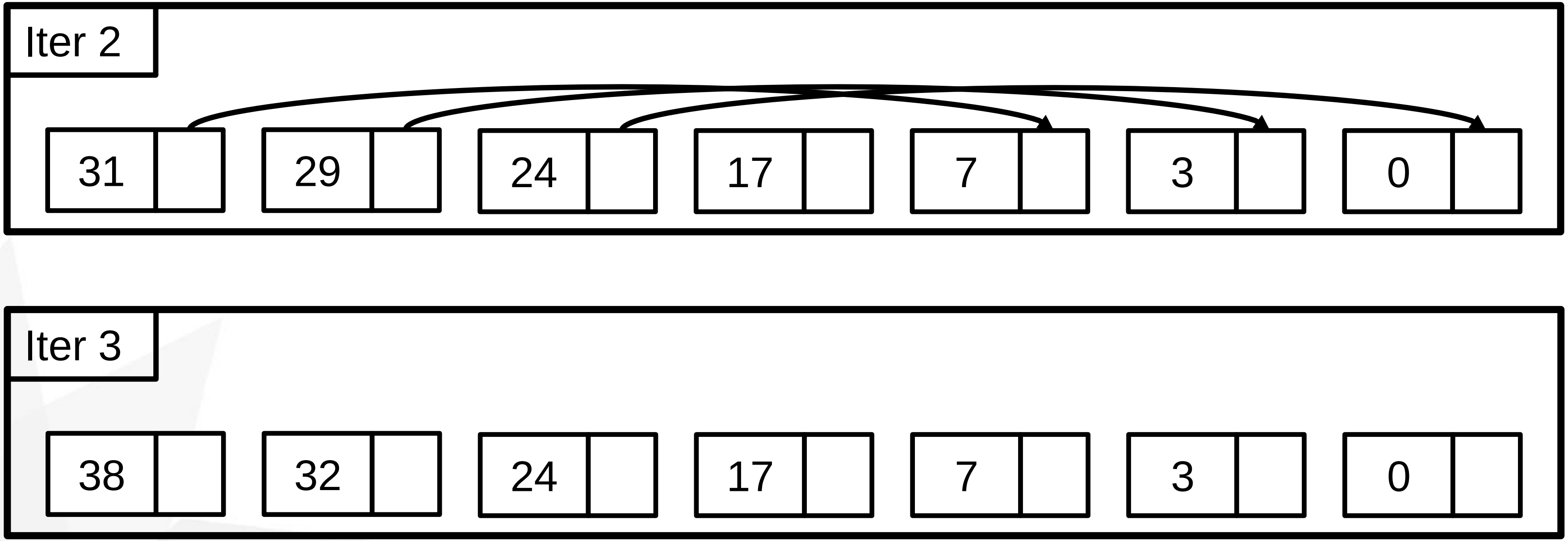
Min-Plus Matrix Multiplication



Path doubling algorithm(or Path jumping) is an algorithm that can coumpte a distance of edge in a logarithmic time.

Background

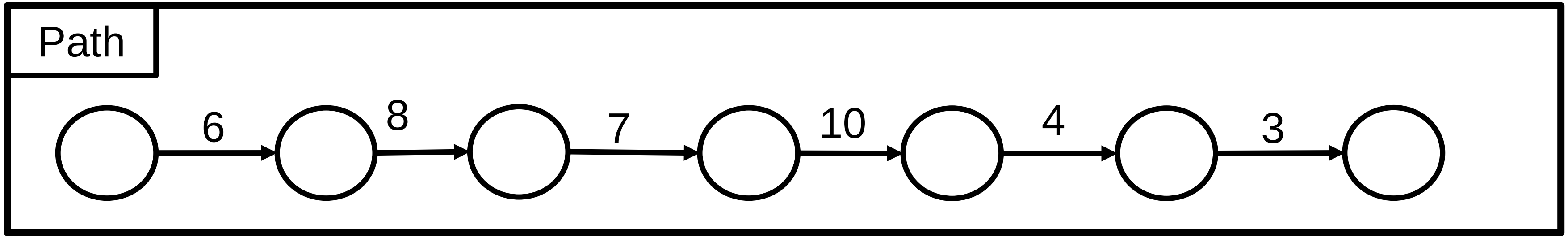
Min-Plus Matrix Multiplication

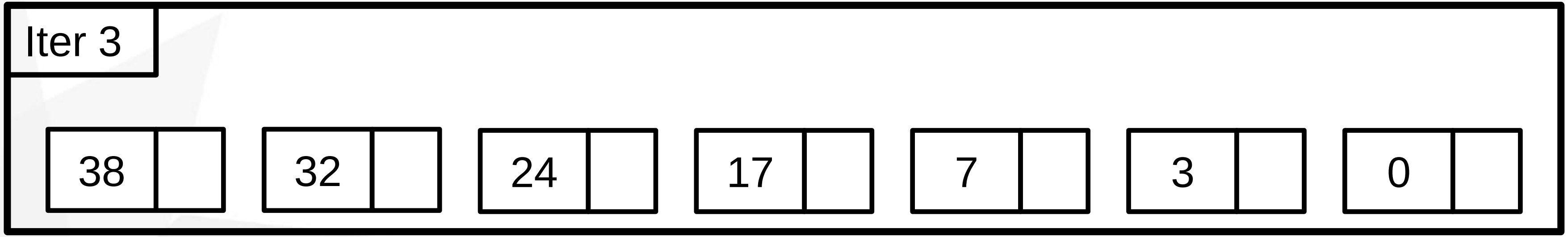


Path doubling algorithm(or Path jumping) is an algorithm that can coumpte a distance of edge in a logarithmic time.

Background

Min-Plus Matrix Multiplication





Path doubling algorithm(or Path jumping) is an algorithm that can coumpte a distance of edge in a logarithmic time.

Background

Min-Plus Matrix Multiplication

Algorithm 1 FLOYD-WARSHALL algorithm for APSP

```
function FLOYDWARSHALL(G = (V, E)):
    Let n ← dim(V)
    Let Dist[i,j] = { w_{i,j}   if(i,j) ∈ E
                    { ∞         otherwise
    for k = {1, 2...,n} do:
        for i = {1, 2...,n} do:
            for j = {1, 2...,n} do:
                Dist[i,j] = min{Dist[i,j], Dist[i,k]+Dist[k,j]}
    Return Dist
```

Focus on the actual algorithm

Figure 12. Floyd-warshall algorithm

Background

Min-Plus Matrix Multiplication

$$
\begin{array}{l}
\textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do:} \\
\quad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do:} \\
\quad\quad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do:} \\
\quad\quad\quad Dist[i, j] = min\{Dist[i, j], Dist[i, k] + Dist[k, j]\}
\end{array}
$$

$$A \oplus B = min(A, B)$$
$$A \otimes B = A + B$$

$$
\begin{array}{l}
\textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do:} \\
\quad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do:} \\
\quad\quad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do:} \\
\quad\quad\quad Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]
\end{array}
$$

Background

Min-Plus Matrix Multiplication

$$
\begin{aligned}
&\textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\qquad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\qquad\qquad Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]
\end{aligned}
$$

$\Downarrow$ Path doubling algorithm

$$
\begin{aligned}
&\textbf{for } l = \{1, 2, \cdots, logn\} \textbf{ do:}\\
&\qquad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\qquad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\qquad\qquad \textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\qquad\qquad Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]
\end{aligned}
$$

Background

Min-Plus Matrix Multiplication

It forms a semi-ring

$$A \oplus B = min(A, B)$$
$$A \otimes B = A + B$$

Path doubling approach

Same form with the matrix multiplication

$$
\begin{aligned}
&\textbf{for } l = \{1, 2, \cdots, logn\} \textbf{ do:}\\
&\quad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\quad \textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do:}\\
&\qquad\quad Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]
\end{aligned}
$$

Check for every intermediaries

Background

Min-Plus Matrix Multiplication

$$
\begin{aligned}
&\textbf{for } l = \{1, 2, \cdots, logn\} \textbf{ do}: \\
&\quad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do}: \\
&\qquad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do}: \\
&\qquad\quad \textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do}: \\
&\qquad\quad Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]
\end{aligned}
$$

$$\Downarrow$$

$$
\begin{aligned}
&\textbf{for } l = \{1, 2, \cdots, logn\} \textbf{ do}: \\
&\quad Dist = Dist \oplus Dist \otimes Dist
\end{aligned}
$$

Background

Min-Plus Matrix Multiplication

$$\textbf{for } l = \{1, 2, \cdots, logn\} \textbf{ do:}$$
$$Dist = Dist \oplus Dist \otimes Dist$$

Since Mathematical structure with $\oplus$, $\otimes$ is a semiring.
There are some GPU-implemented for this kinds of approach. (level-3 BLAS)

There is an another approach to solve this.
There are many matrix multiplication algorithms.
There are some GPU implemented matrix multiplication algorithm either.
It takes $O(|V|^3 log|V|)$, but there is a faster way to calculate the matrix.

Background

Min-Plus Matrix Multiplication

$$C = AB \text{ and } C, A, B \in R^{2^n \times 2^n}$$

$$A = \begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \; B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} \; C = \begin{bmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{bmatrix}$$

$$M_1 = (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}),\ M_2 = (A_{2,1} + A_{2,2})B_{1,1}$$

$$M_3 = A_{1,1}(B_{1,2} - B_{2,2}),\ M_4 = A_{2,2}(B_{2,1} - B_{1,1})$$

$$M_5 = (A_{1,1} + A_{1,2})B_{2,2},\ M_6 = (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2})$$

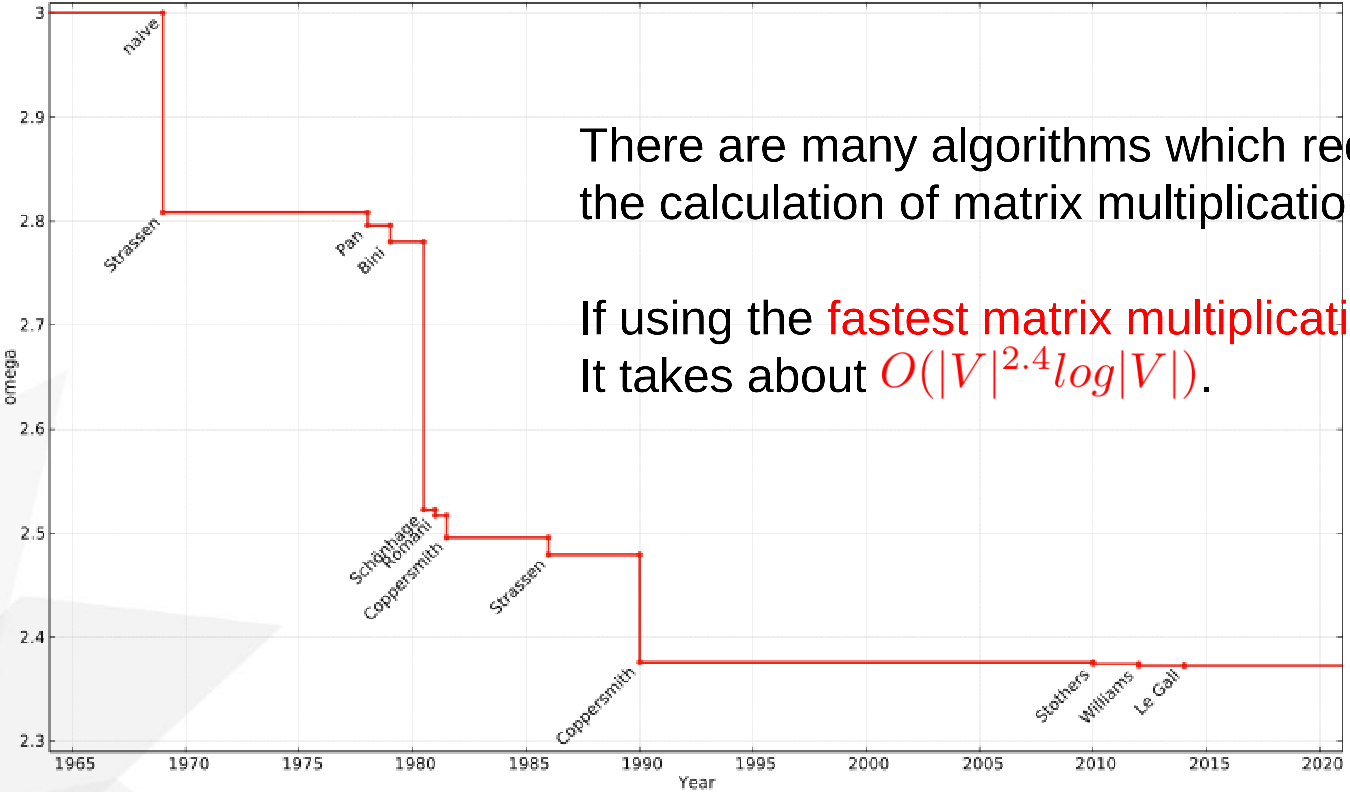
$$M_7 = (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2}),\ C_{1,1} = M_1 + M_4 - M_5 + M_7$$

$$C_{1,2} = M_3 + M_5,\ C_{2,1} = M_2 + M_4,\ C_{2,2} = M_1 - M_2 + M_3 + M_6$$

This is a Strassen algorithm which takes $O(|V|^{2.8} log|V|)$.
Which reduce calculation for matrix multiplication.
GPU implementation provided at PPoPP 2011

Background

Min-Plus Matrix Multiplication



There are many algorithms which reducing the calculation of matrix multiplication.

If using the fastest matrix multiplication,
It takes about $O(|V|^{2.4} log|V|)$.

Figure 14. Time complexity of matrix multiplication algorithms

Background

Blocked Floyd-Warshall algorithm

Algorithm 2 A blocked version of Floyd-Warshall algorithm for Apsp

---

function BlockedFloydWarshall(A):
    for $k = \{1, 2 \ldots, n_b\}$ **do**:
Diagonal Update
        $A(k,k) \leftarrow$ Floyd-Warshall$(A(k,k))$
Panel Update
        $A(k,:) \leftarrow A(k,:) \bigoplus A(k,k) \otimes A(k,:)$
        $A(:,k) \leftarrow A(:,k) \bigoplus A(:,k) \otimes A(k,k)$
MinPlus Outer Product
        for $i = \{1, 2 \ldots, n_b\}, i \neq k$ **do**:
            for $j = \{1, 2 \ldots, n_b\}, j \neq k$ **do**:
                $A(i,j) \leftarrow A(i,j) \bigoplus A(i,k) \otimes A(k,j)$
    Return A

Figure 15. A blocked version of Floyd-Warshall algorithm

Blocked Floyd-Warshall based on Floyd-Warshall as it can be known from the name.

It divides a result matrix to blocks and calculates in local.

This algorithm originally intended to reduce the communication cost between a memory and a cache.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	X	1	2	3	4	5	6	7
2	1	X	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

Blocked Floyd-Warshall based on Floyd-Warshall as it can be known from the name.

It divides a result matrix to blocks and calculates in local.

This algorithm originally intended to reduce the communication cost between a memory and a cache.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	X	1	2	3	4	5	6	7
2	1	X	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

Blocked Floyd-Warshall based on Floyd-Warshall as it can be known from the name.

It divides a result matrix to blocks and calculates in local.

This algorithm originally intended to reduce the communication cost between a memory and a cache.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 3~4 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 3~4 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

K = 3~4 (At the view of classic Floyd-Warshall algorithm)

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

K = 5~6 (At the view of classic Floyd-Warshall algorithm)

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

K = 5~6 (At the view of classic Floyd-Warshall algorithm)

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 5~6 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

K = 7~8 (At the view of classic Floyd-Warshall algorithm)

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

K = 7~8 (At the view of classic Floyd-Warshall algorithm)

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

K = 7~8 (At the view of classic Floyd-Warshall algorithm)

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

$$Dist[i,j] = min(Dist[i,j], Dist[i,k] + Dist[k,j])$$

Yellow block calculating over sky block and itself.

Stage 1: Self-dependent block.
It proceed Floyd-Warshall in itself.
Stage 2: It proceeds with same line number of column blocks or row blocks.
Stage 3: It proceeds with lefts.

Background

Blocked Floyd-Warshall algorithm

$$
\begin{aligned}
&\textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do}: \\
&\qquad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do}: \\
&\qquad\qquad \textbf{for } j = \{1, 2, \cdots, n\} \textbf{ do}: \\
&\qquad\qquad\qquad Dist[i, j] = min\{Dist[i, j], Dist[i, k] + Dist[k, j]\}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{for } k = \{1, 2, \cdots, n_b\} \textbf{ do}: \\
&\qquad Dist[k, k] = Floyd - Warshall(Dist[k, k]) \\
&\qquad Dist[k, ;] = Dist[k, ;] \oplus Dist[k, k] \otimes Dist[k, ;] \\
&\qquad Dist[;, k] = Dist[; k] \oplus Dist[;, k] \otimes Dist[k, k] \\
&\qquad \textbf{for } i = \{1, 2, \cdots, n_b\}, j = \{1, 2, \cdots, n_b\}, i \neq k, j \neq k \textbf{ do}: \\
&\qquad\qquad Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]
\end{aligned}
$$

Background

Blocked Floyd-Warshall algorithm

$\textbf{for } k = \{1, 2, \cdots, n\} \textbf{ do:}$

$\quad \textbf{for } i = \{1, 2, \cdots, n\} \textbf{ do:}$

$\quad\quad \textbf{r } j = \{1, 2, \cdots, n\} \textbf{ do:}$

$\quad\quad\quad Dist[$

Synchronized result at the end of same intermediary

$\textbf{for } k = \{1, 2, \cdots, n_b\} \textbf{ do:}$

$$Dist[k, k] = Floyd - Warshall(Dist[k, k])$$

$$Dist[k, ;] = Dist[k, ;] \oplus Dist[k, k] \otimes Dist[k, ;]$$

$$Dist[;, k] = Dist[; k] \oplus Dist[;, k] \otimes Dist[k, k]$$

$\textbf{for } i = \{1, 2, \cdots, n_b\}, j = \{1, 2, \cdots, n_b\}, i \neq k, j \neq k \textbf{ do:}$

$$Dist[i, j] = Dist[i, j] \oplus Dist[i, k] \otimes Dist[k, j]$$

Background

Blocked Floyd-Warshall algorithm

Distance don't increase with iteration

$$\boxed{Dist[i,j]} = \boxed{min\{}Dist[i,j], Dist[i,k] + Dist[k,j]\}$$

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

For iteration of stage 1 in yellow block can be proceed values in it.

All the other values are out of mind.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

For iteration of stage 2 in yellow block can be proceed values in yellow and blue blocks.
All the other values are out of mind.

Every values in the blue block is newer than original Floyd-warshall algorithm.
Therefore, it finds at least better path than original.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	X	3	4	5	6	7
4	3	3	3	X	4	5	6	7
5	4	4	4	4	X	5	6	7
6	5	5	5	5	5	X	6	7
7	6	6	6	6	6	6	X	7
8	7	7	7	7	7	7	7	X

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

For iteration of stage 2 in yellow block can be proceed values in yellow and blue blocks.
All the other values are out of mind.

Every values in the blue block is newer than original Floyd-warshall algorithm. Therefore, it finds at least better path than original.

Background

Blocked Floyd-Warshall algorithm

	1	2	3	4	5	6	7	8
1	2	1	2	3	4	5	6	7
2	1	2	2	3	4	5	6	7
3	2	2	4	3	4	5	6	7
4	3	3	3	6	4	5	6	7
5	4	4	4	4	8	5	6	7
6	5	5	5	5	5	10	6	7
7	6	6	6	6	6	6	12	7
8	7	7	7	7	7	7	7	14

K = 1~2 (At the view of classic Floyd-Warshall algorithm)

$$\boxed{Dist[i,j]} = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

For iteration of stage 3 in yellow block can be proceed values in yellow and blue blocks.
All the other values are out of mind.

Every values in the blue block is newer than original Floyd-warshall algorithm. Therefore, it finds at least better path than original.

Background

Blocked Floyd-Warshall algorithm

Algorithm 2 A blocked version of FLOYD-WARSHALL algorithm for APSP

```
function BLOCKEDFLOYDWARSHALL(A):
    for k = {1, 2...,n_b} do:
  Diagonal Update
        A(k,k) ← FLOYD-WARSHALL(A(k,k))
  Panel Update
        A(k,:) ← A(k,:) ⊕ A(k,k) ⊗ A(k,:)
        A(:,k) ← A(:,k) ⊕ A(:,k) ⊗ A(k,k)
  MinPlus Outer Product
        for i = {1, 2...,n_b}, i ≠ k do:
            for j = {1, 2...,n_b}, j ≠ k do:
                A(i,j) ← A(i,j) ⊕ A(i,k) ⊗ A(k,j)
    Return A
```

Figure 15. A blocked version of Floyd-Warshall algorithm

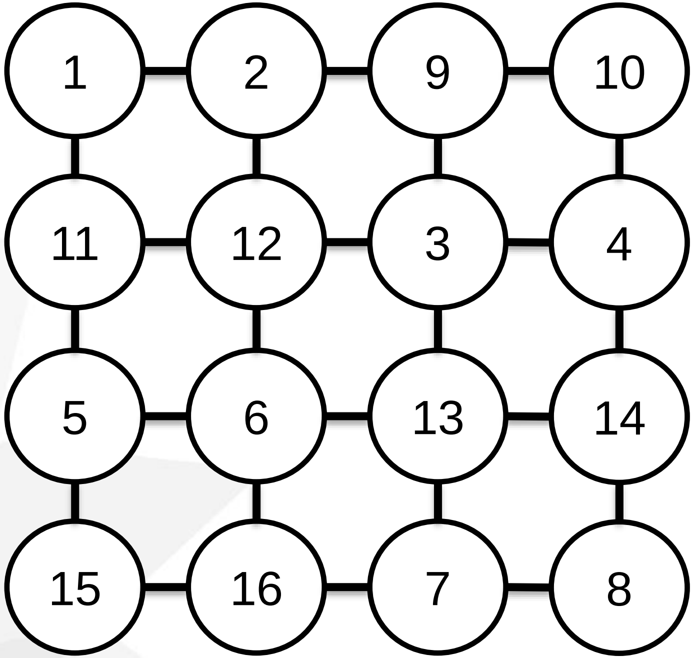
Changing computation order reduces the communication cost.

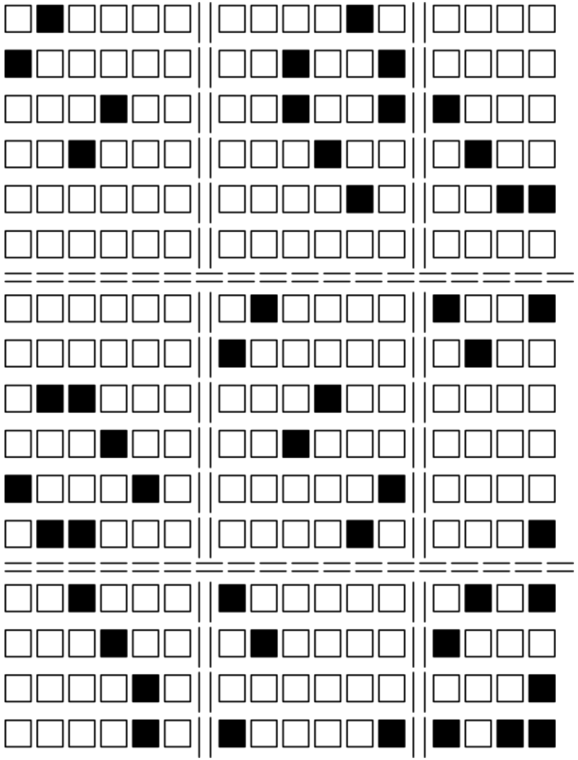
It's the idea.
How we can use this to make more speedup?

Supernodal Floyd-Warshall

Vertex reordering

Let's assume that block size is 6 and given graph is below.

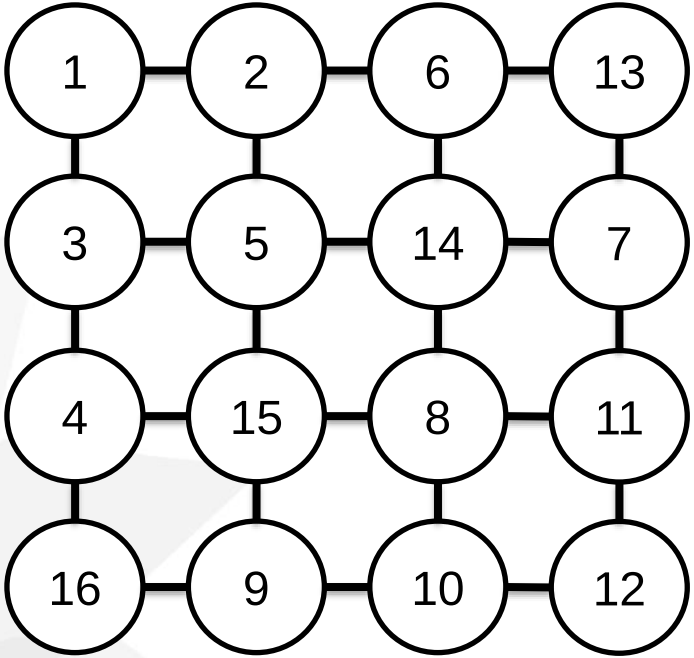


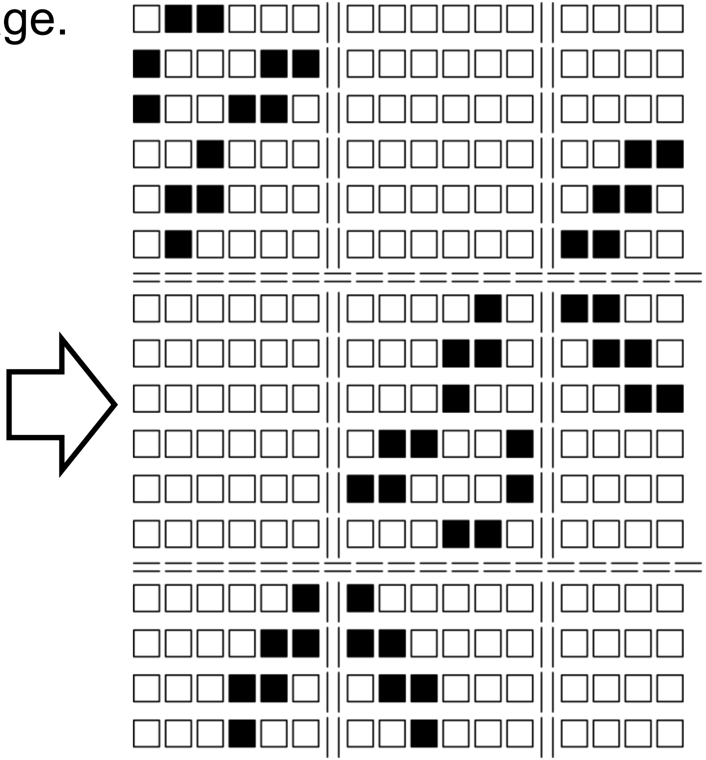


Supernodal Floyd-Warshall

Vertex reordering

It’s identical graph with previous page.
It just reordered vertices.





Supernodal Floyd-Warshall

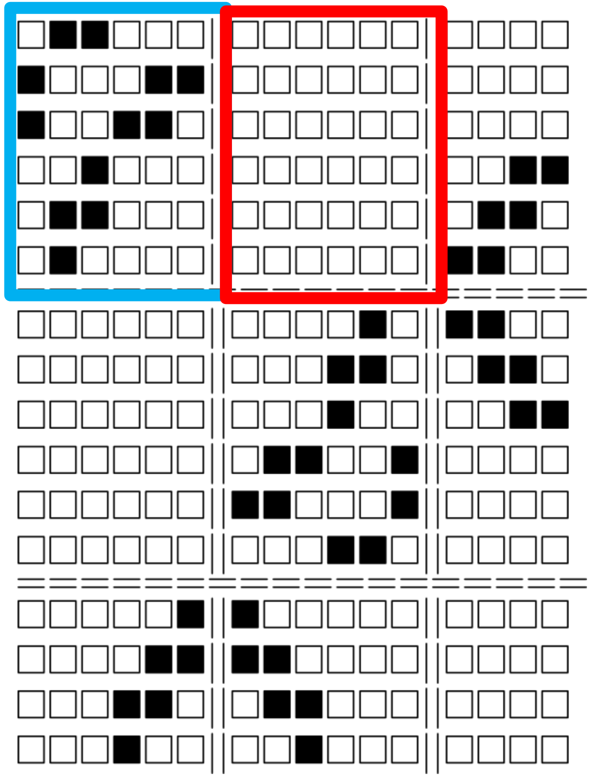
Vertex reordering

Computation is useless at stage 2 over empty block

This an infinity.

$$Dist[i,j] = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

This an infinity.



Supernodal Floyd-Warshall

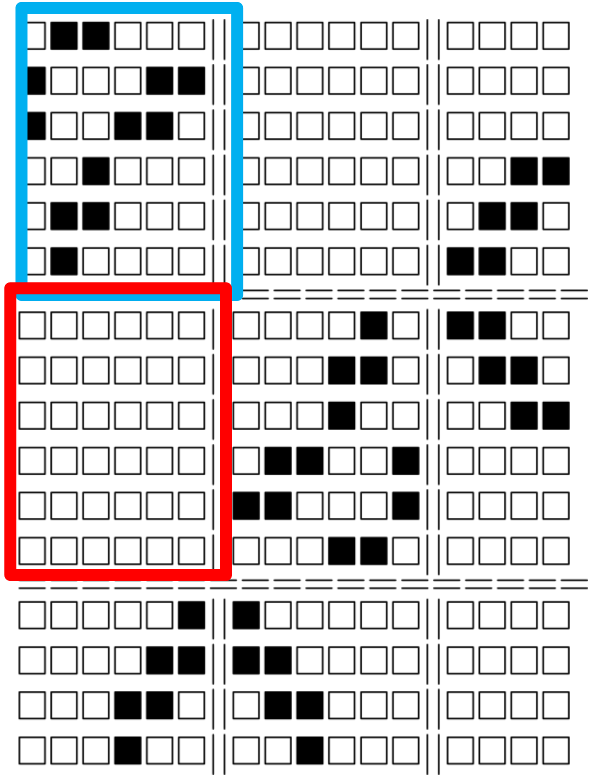
Vertex reordering

Computation is useless at stage 2 over empty block

This an infinity.

$$Dist[i,j] = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

This an infinity.



Supernodal Floyd-Warshall

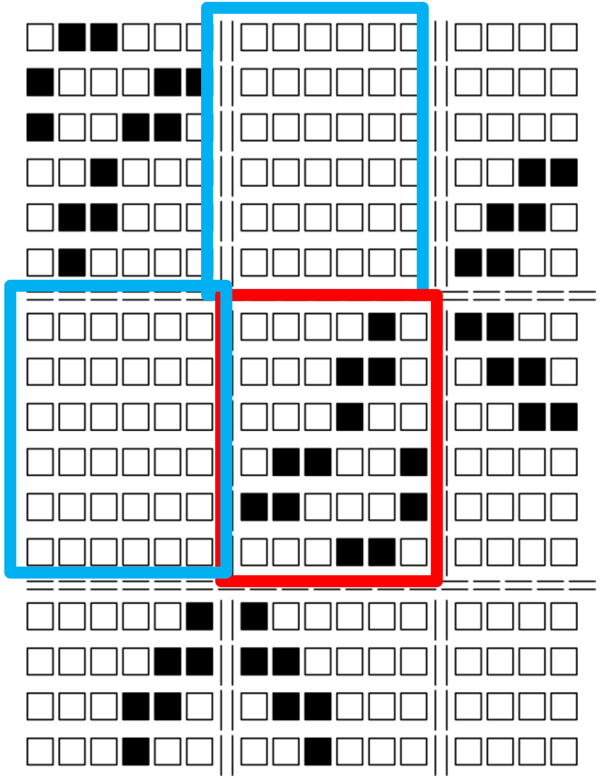
Vertex reordering

Computation is useless at stage 3 over empty block

This an infinity.

$$Dist[i,j] = min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

This an infinity.



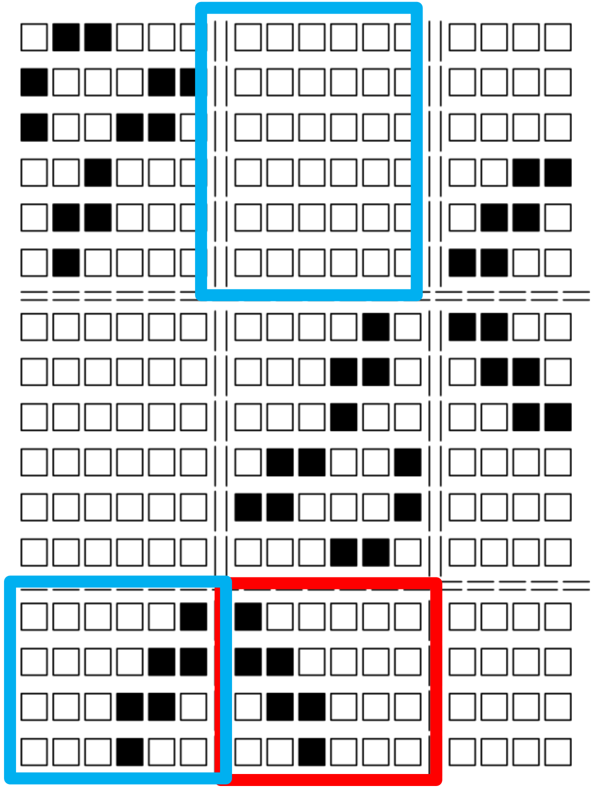
Supernodal Floyd-Warshall

Vertex reordering

Computation is useless at stage 3 over empty block

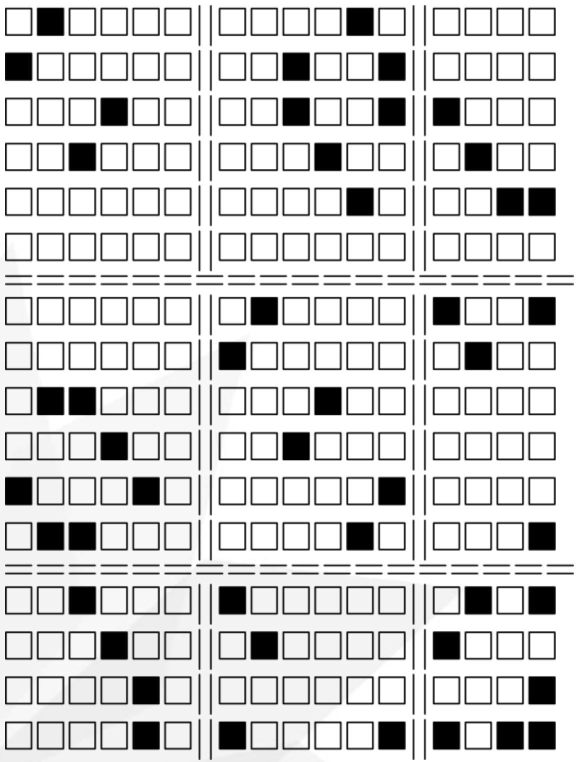
$$Dist[i,j] = \min(\boxed{Dist[i,j]}, \boxed{Dist[i,k]} + \boxed{Dist[k,j]})$$

This an infinity.



Supernodal Floyd-Warshall

Vertex reordering

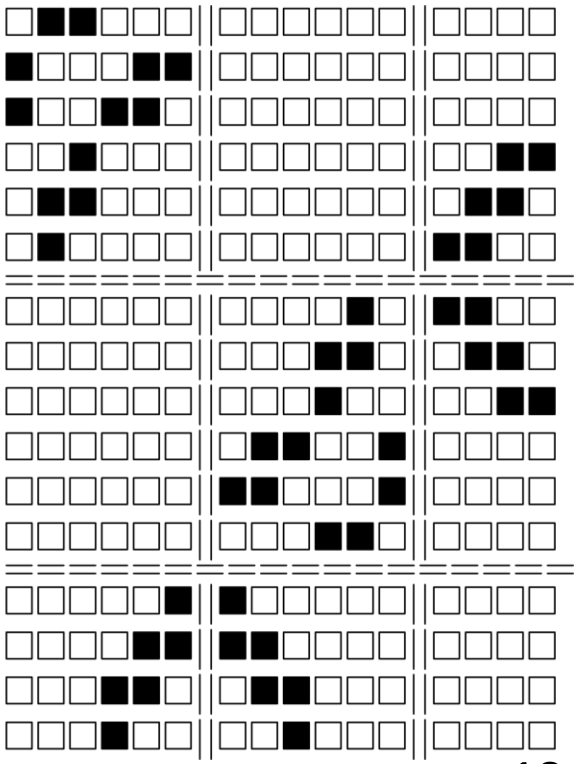


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

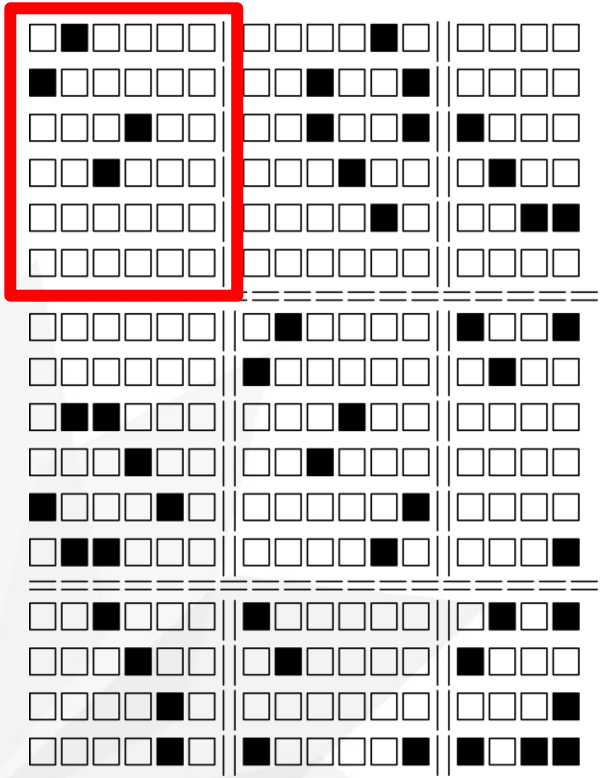
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

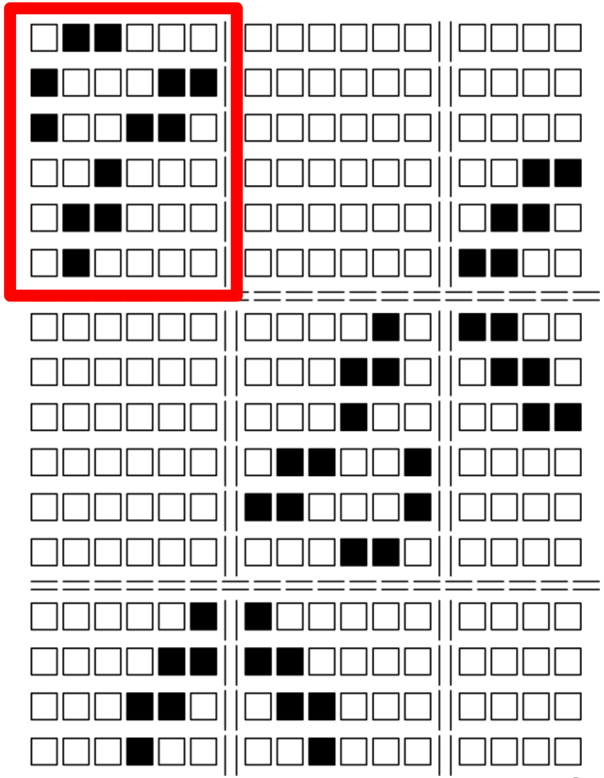


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warsha R it L

Stage 2: It proceed with same line number of columns or rows.

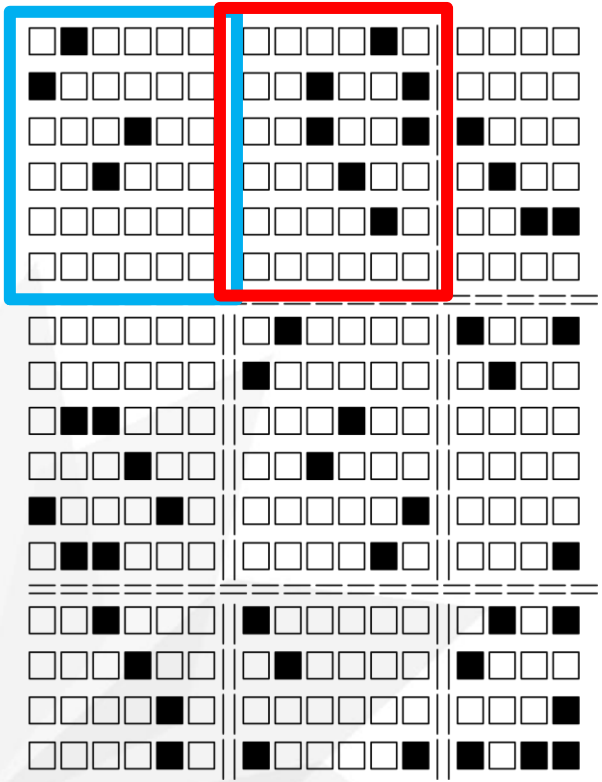
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



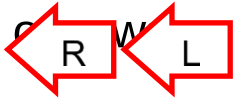
Supernodal Floyd-Warshall

Vertex reordering



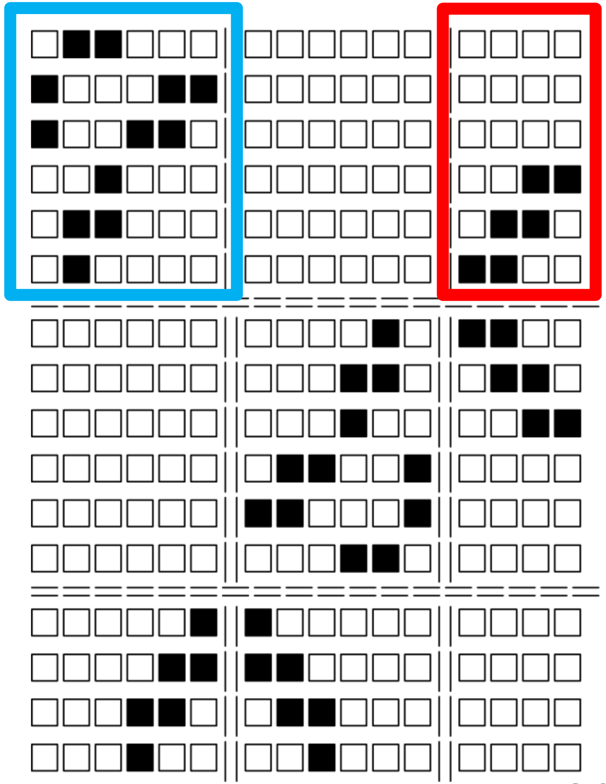
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns o… w



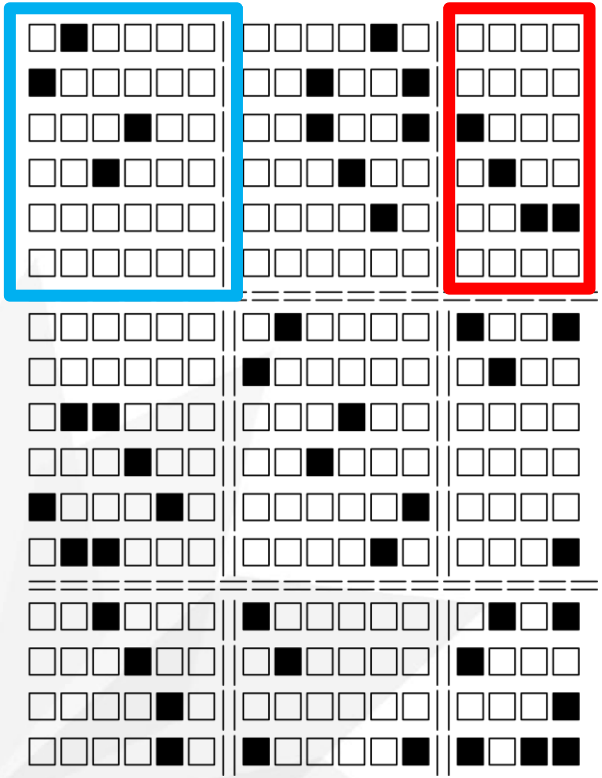
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



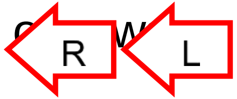
Supernodal Floyd-Warshall

Vertex reordering



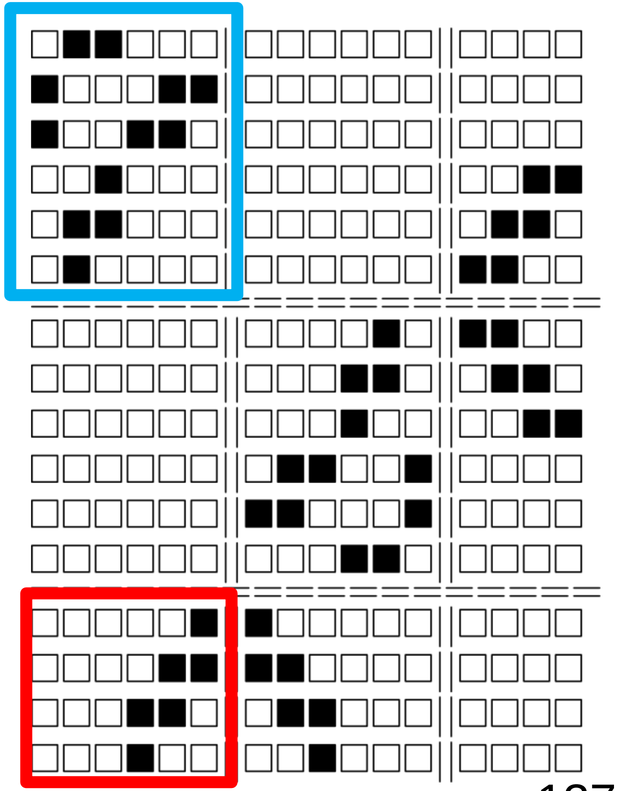
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns of rows.



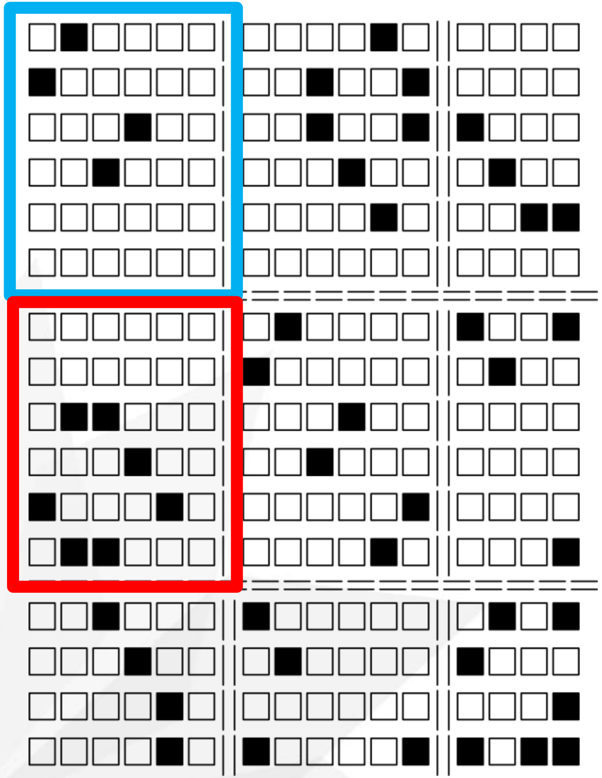
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



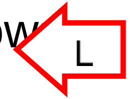
Supernodal Floyd-Warshall

Vertex reordering

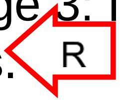


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

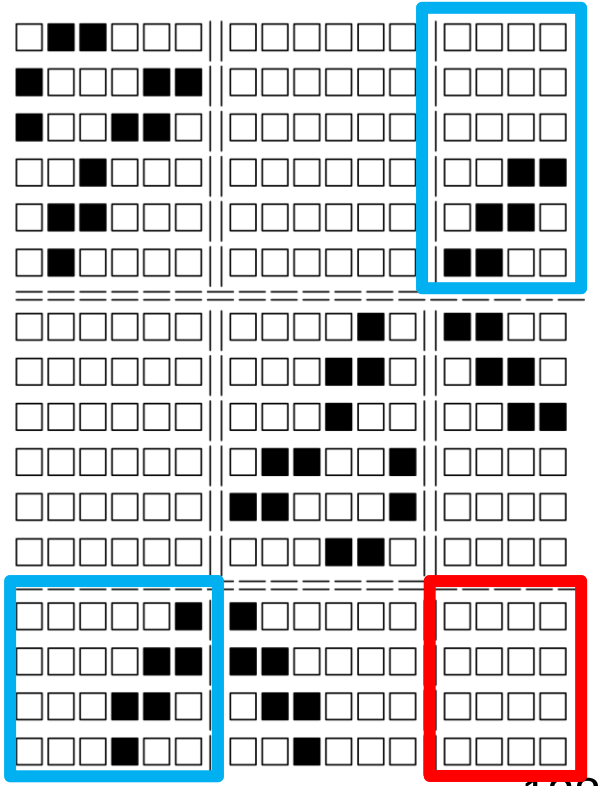
Stage 2: It proceed with same line number of columns or rows.



Stage 3: It proceed with lefts.

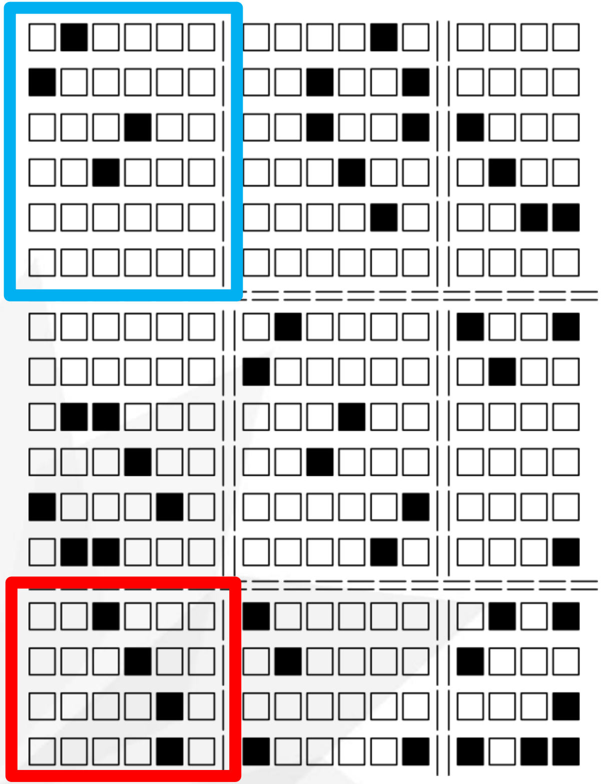


Keep doing it with all value of k.

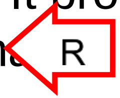


Supernodal Floyd-Warshall

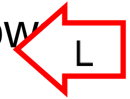
Vertex reordering



Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall itself.

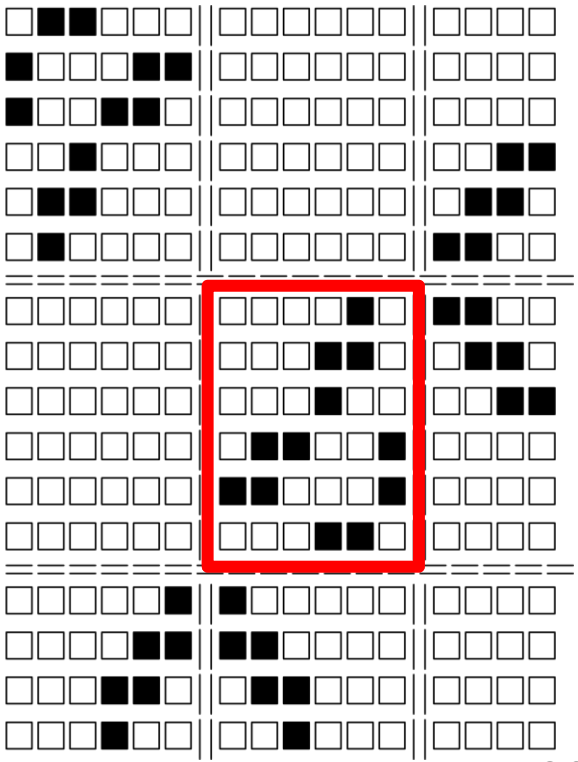


Stage 2: It proceed with same line number of columns or rows.



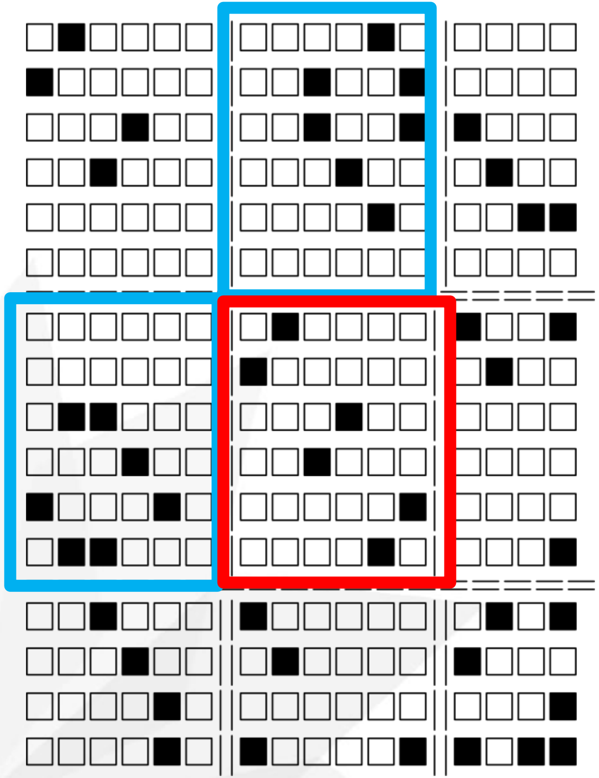
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

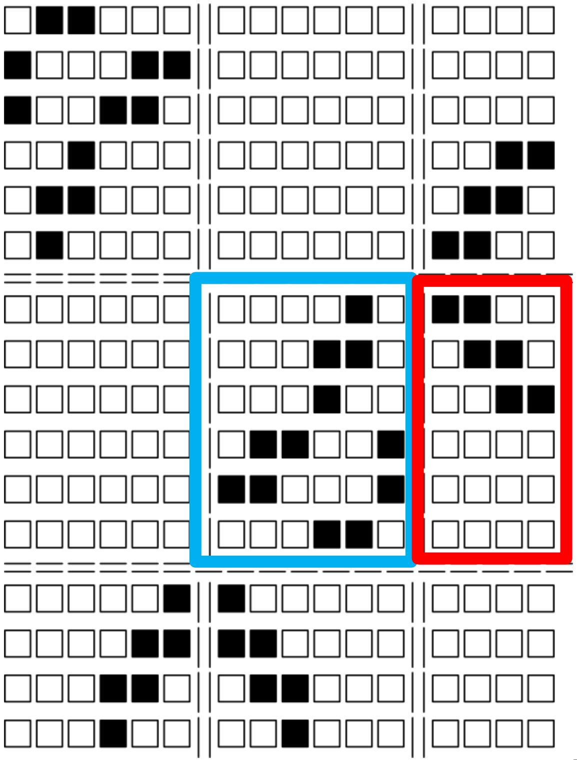


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows. R

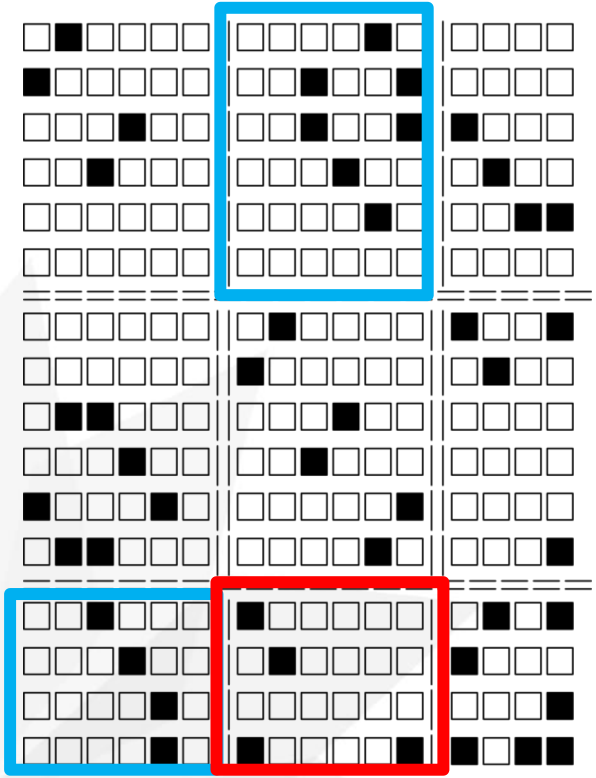
Stage 3: It proceed with lefts. L

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

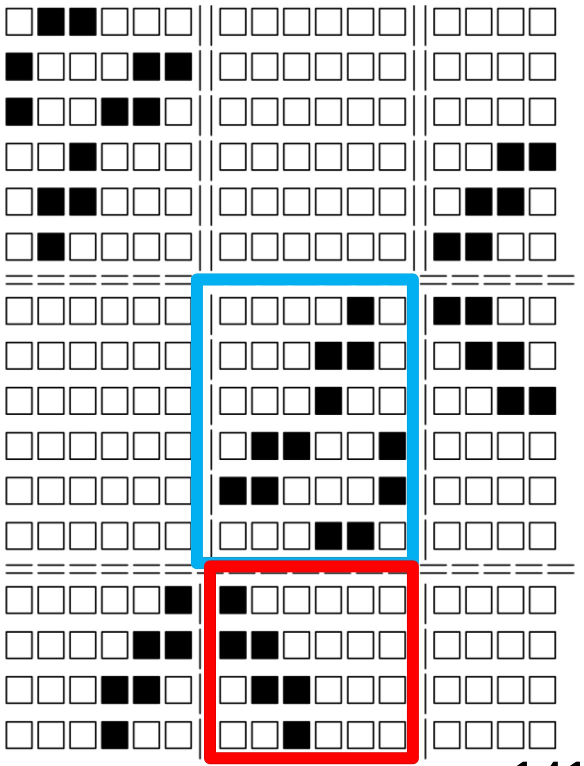


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

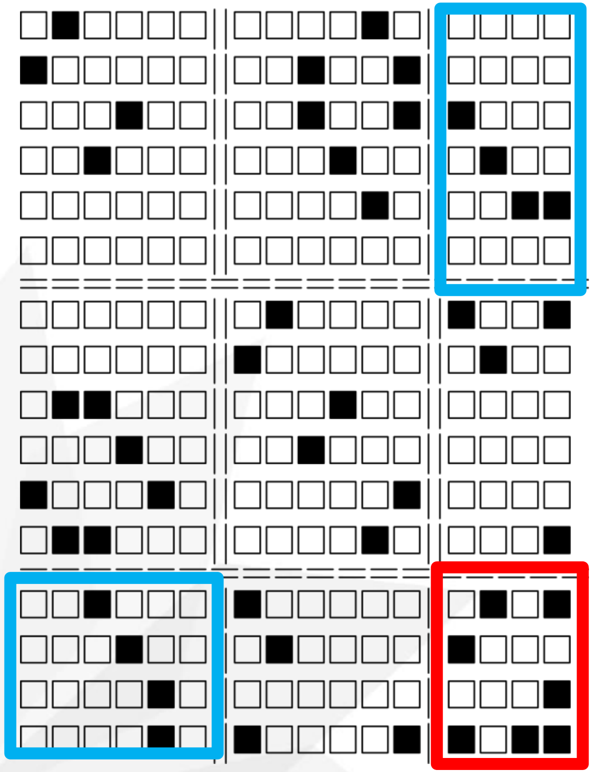
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

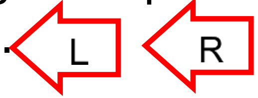
Vertex reordering



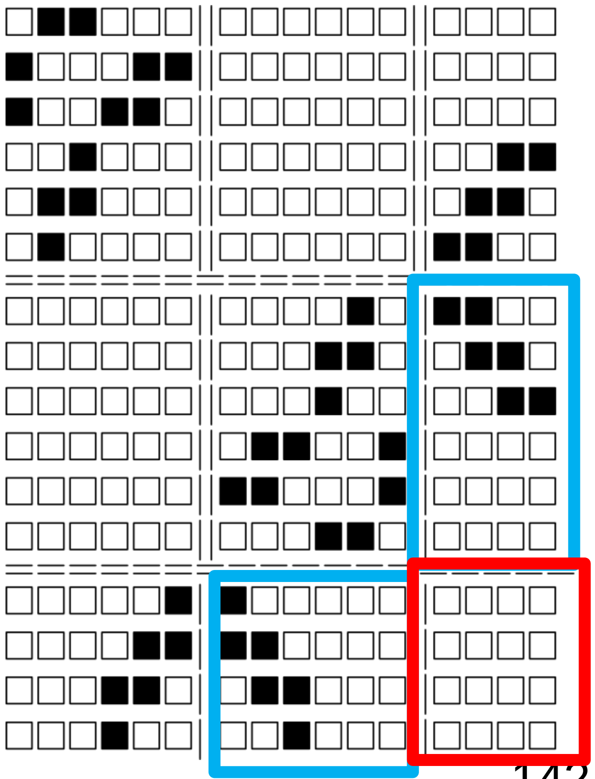
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

Stage 3: It proceed with lefts.

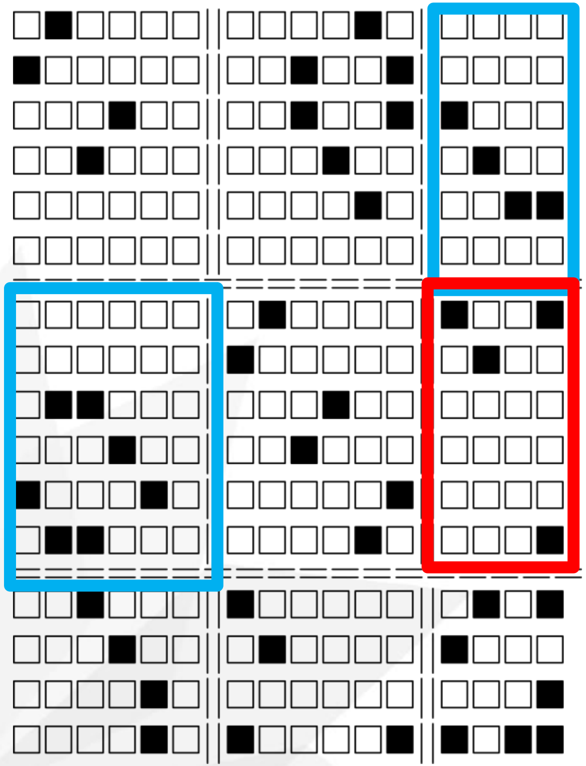


Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

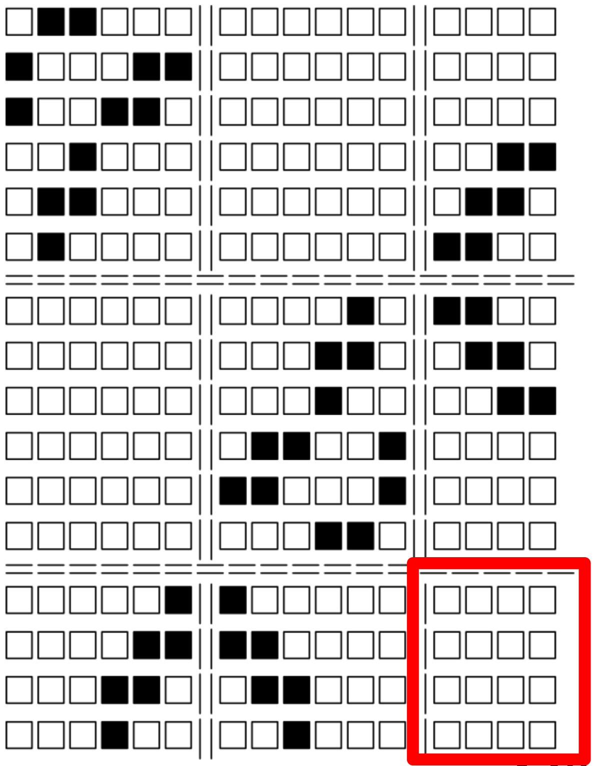


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall itself.

Stage 2: It proceed with same line number of columns or rows.

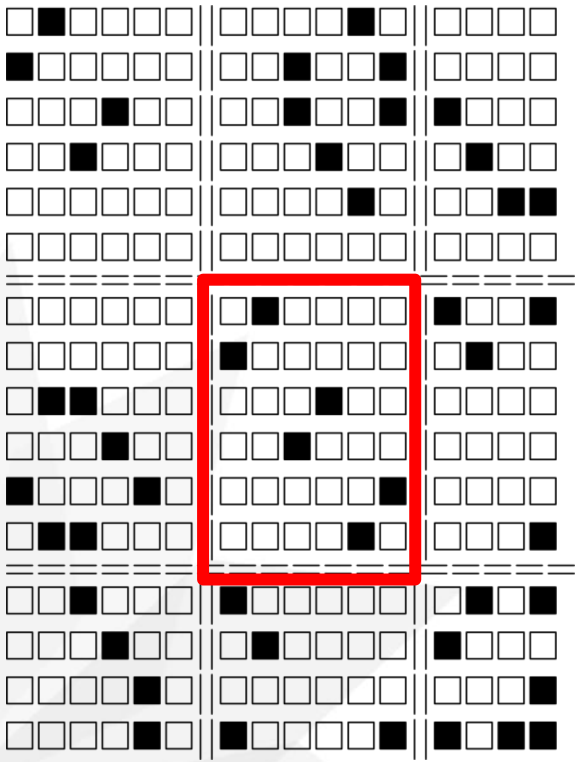
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

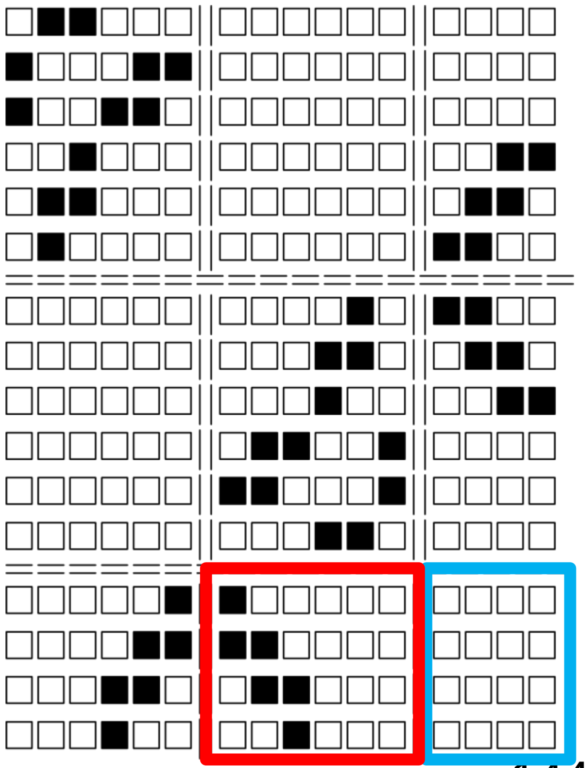


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall itself.

Stage 2: It proceed with same line number of columns or rows

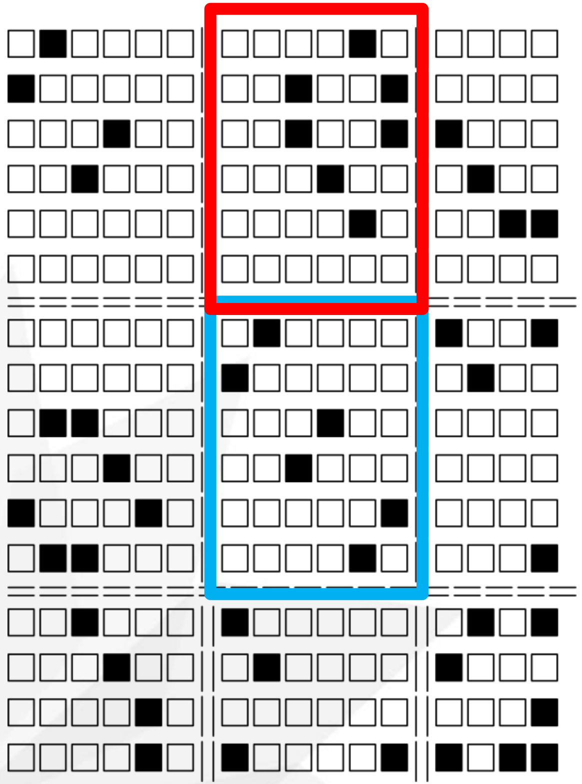
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

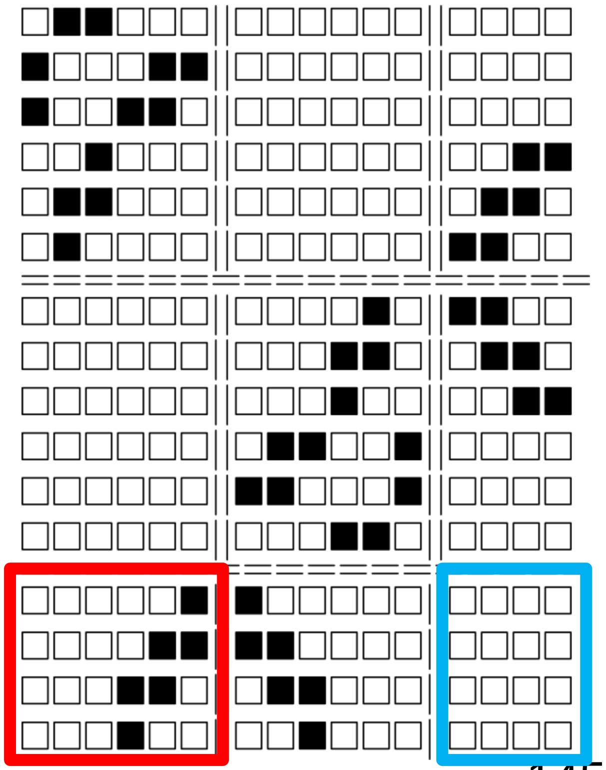


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or ro

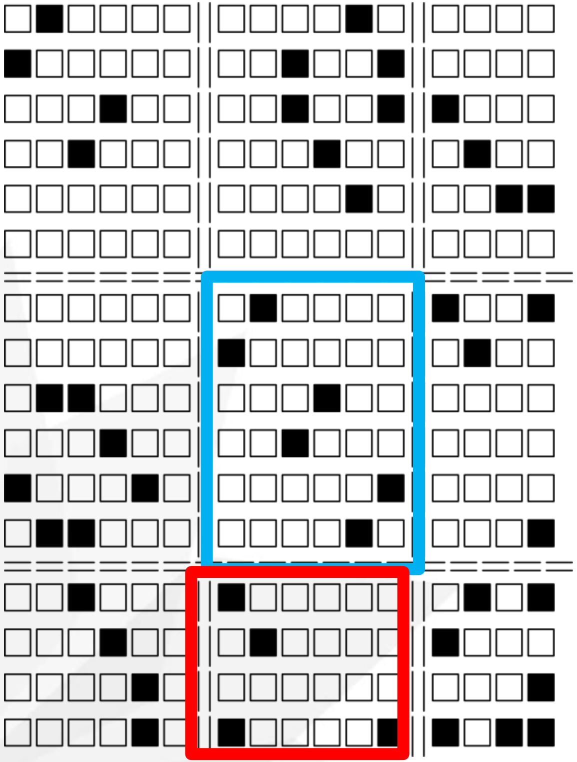
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



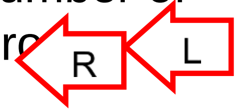
Supernodal Floyd-Warshall

Vertex reordering



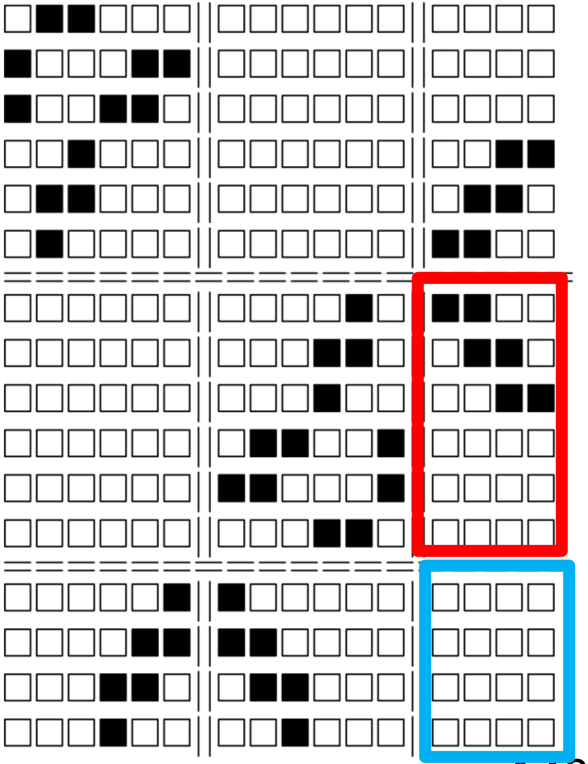
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or r



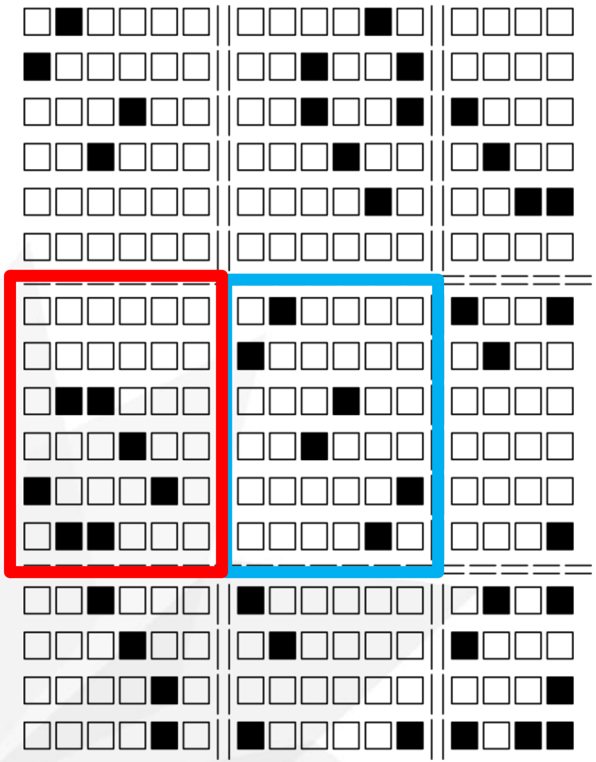
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



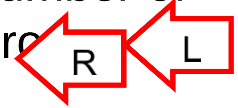
Supernodal Floyd-Warshall

Vertex reordering



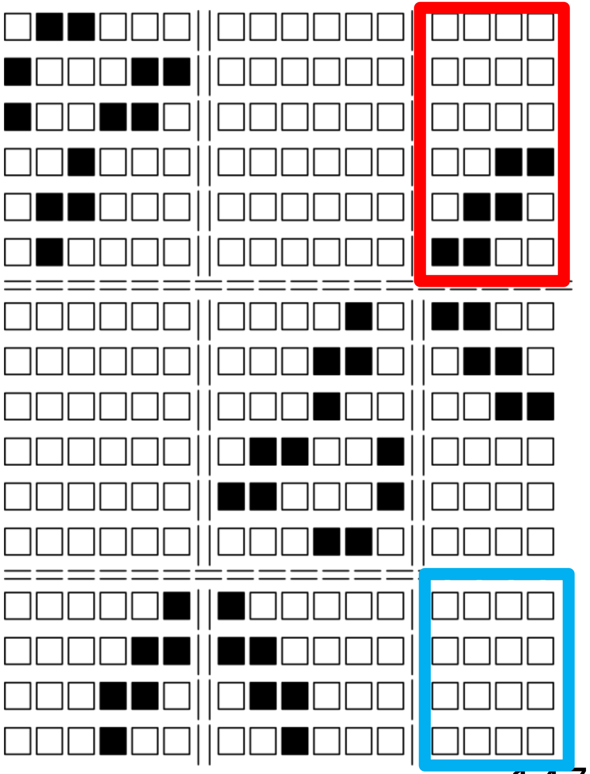
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.



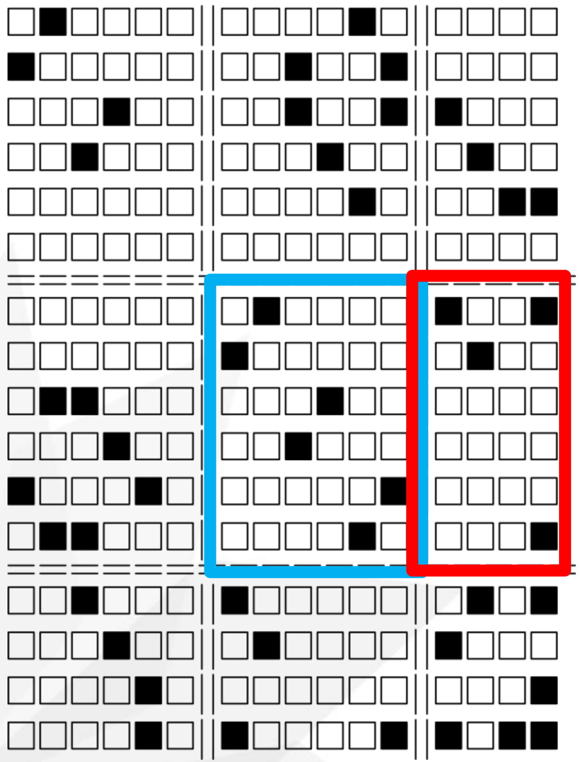
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

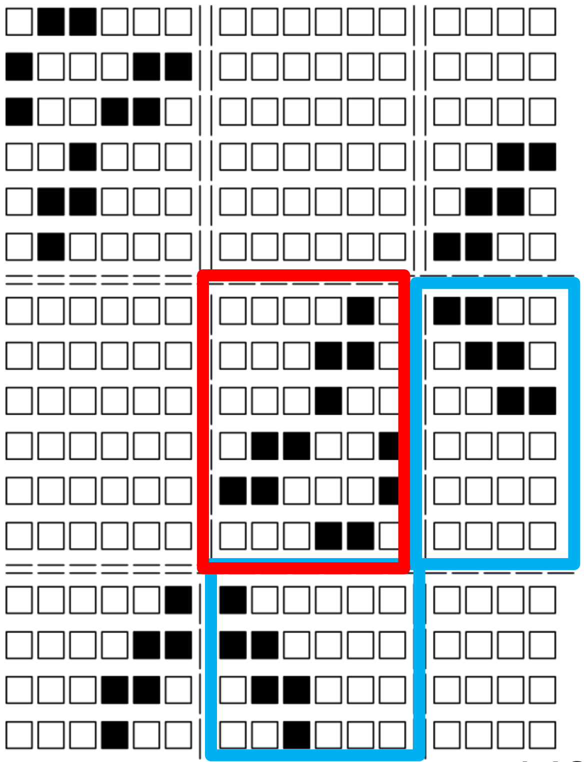


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

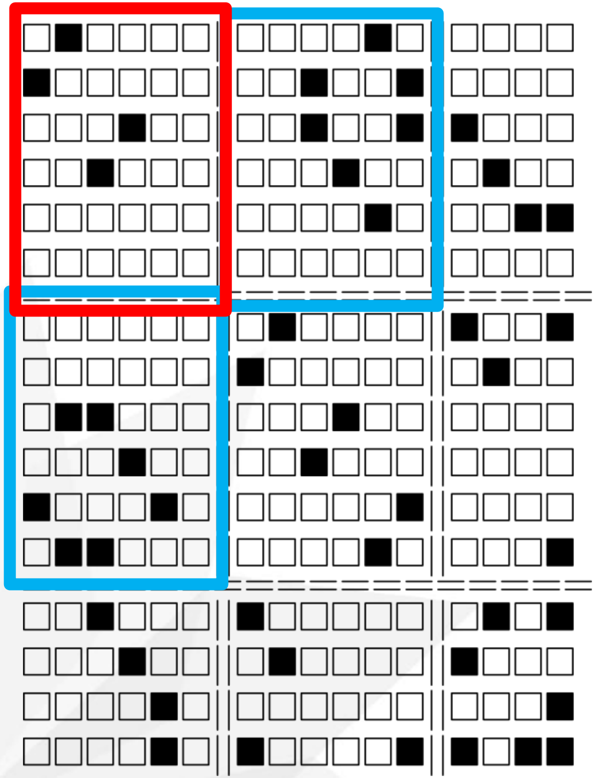
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

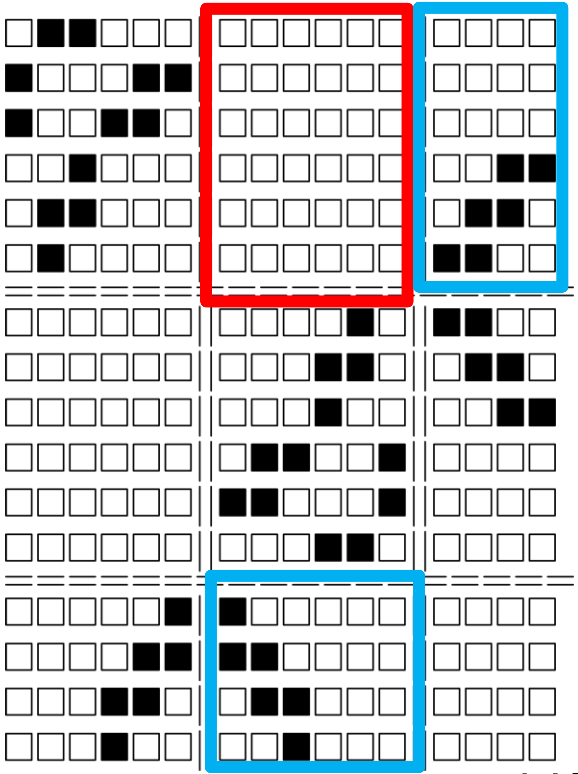


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

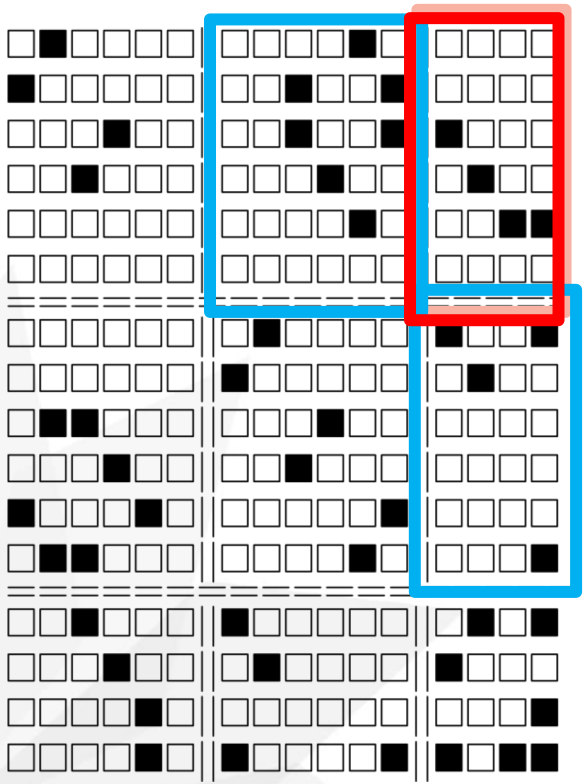
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

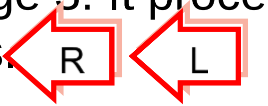
Vertex reordering



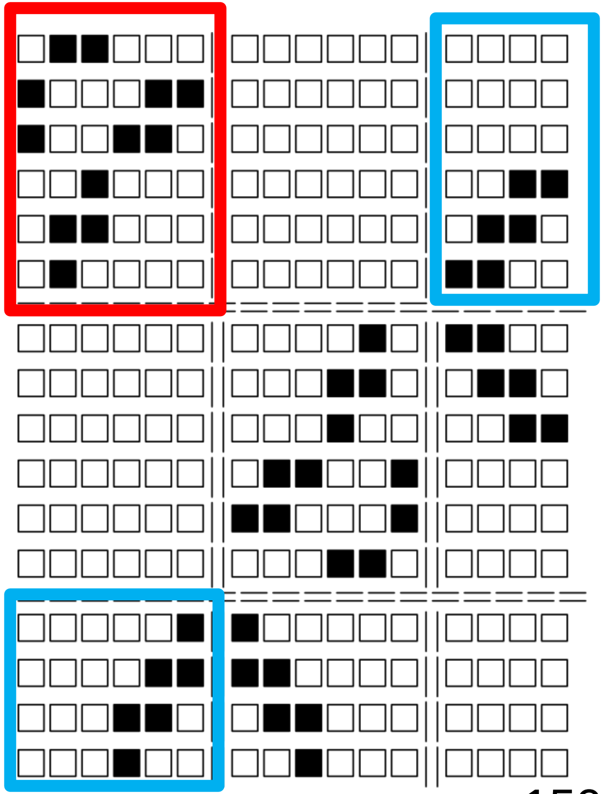
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

Stage 3: It proceed with lefts.

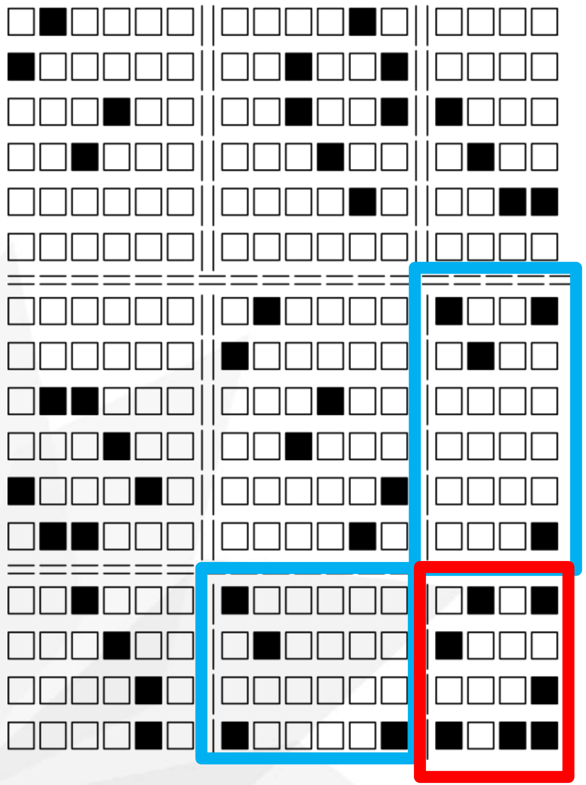


Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

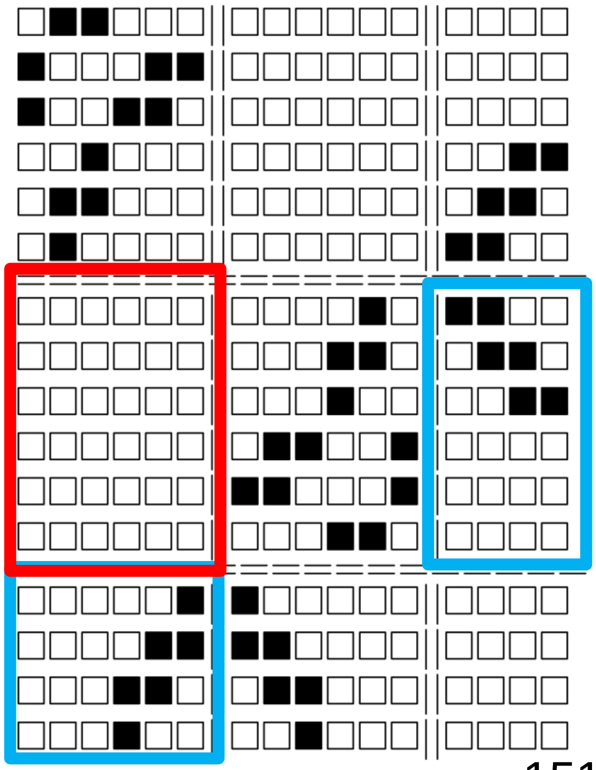


Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

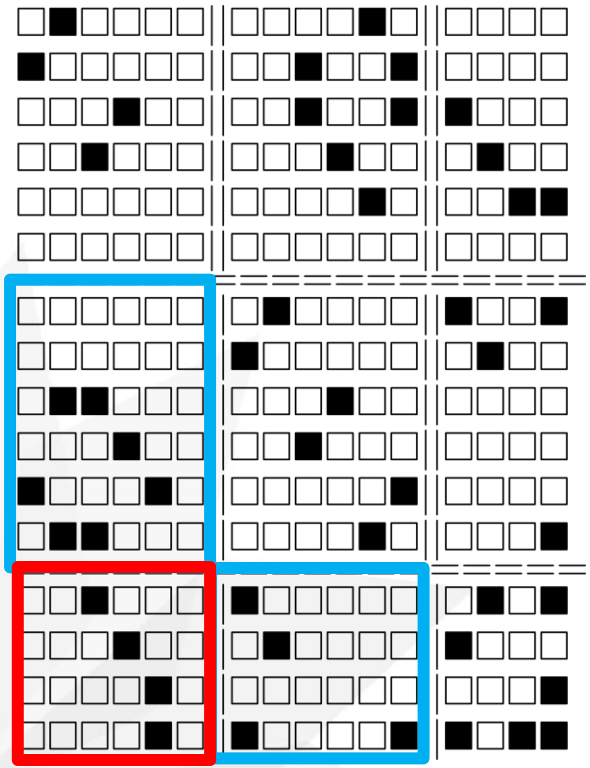
Stage 3: It proceed with lefts.

Keep doing it with all value of k.



Supernodal Floyd-Warshall

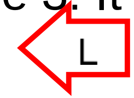
Vertex reordering



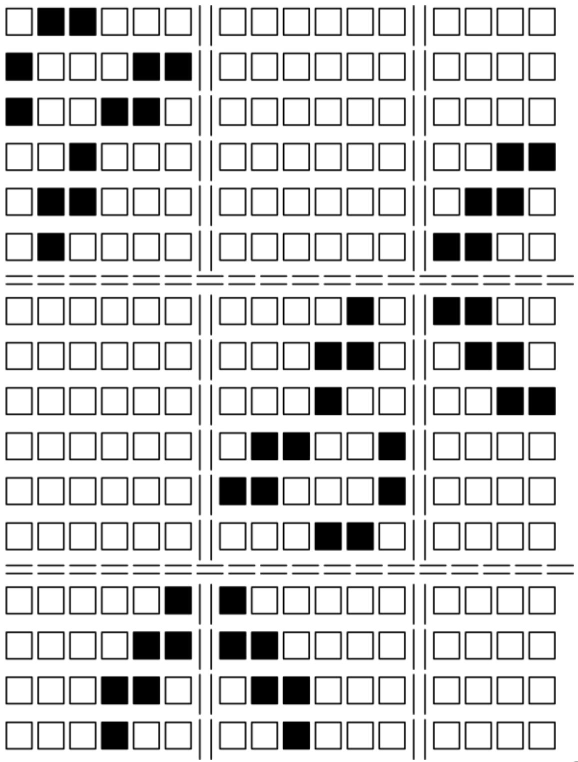
Block-FW algorithm
Stage 1: Self-dependent block. It proceed Floyd-Warshall in itself.

Stage 2: It proceed with same line number of columns or rows.

Stage 3: It proceed with lefts.

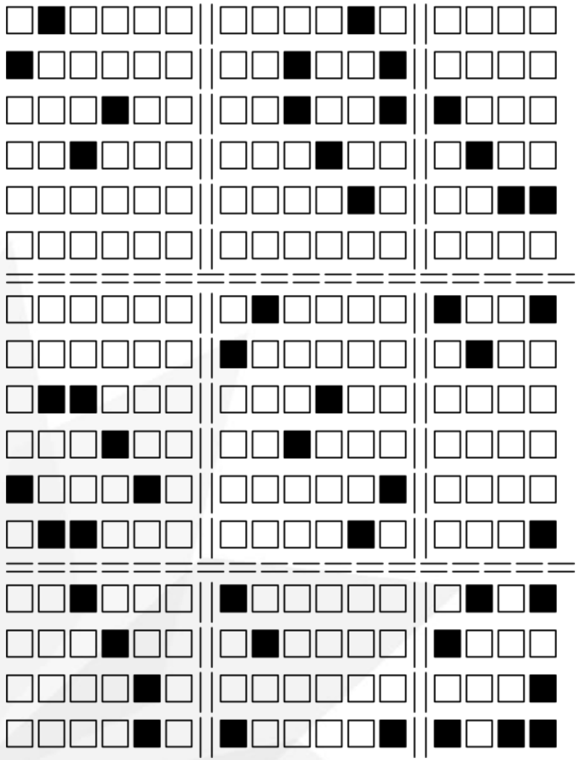


Keep doing it with all value of k.



Supernodal Floyd-Warshall

Vertex reordering

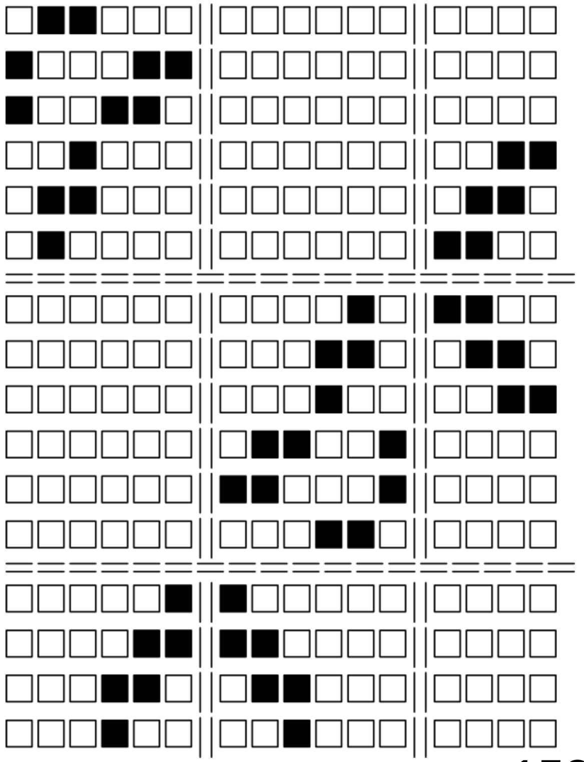


Left graph order requires 27 iteration of the most outer loop.

Right graph order only requires 17 iteration of the most outer loop.

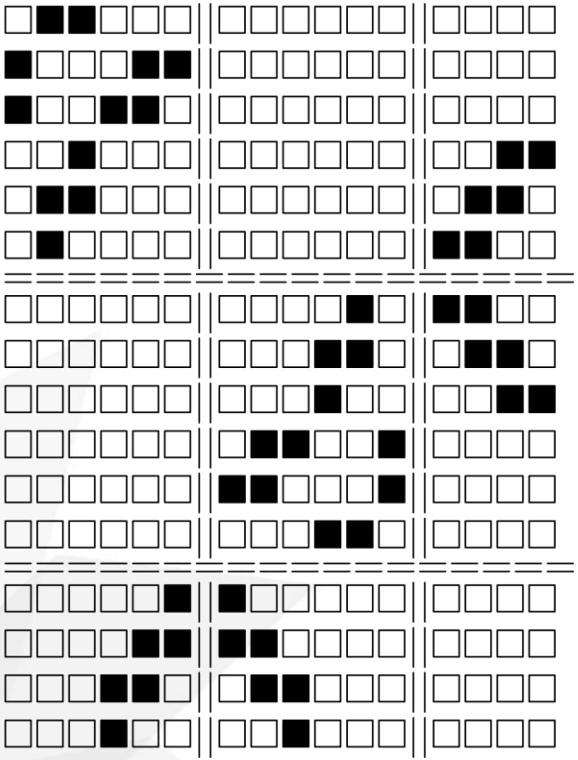
Which means about x1.59 speed-up

How does it work?



Supernodal Floyd-Warshall

Vertex reordering



Stage 1 doesn't ruin the sparsity of matrix.

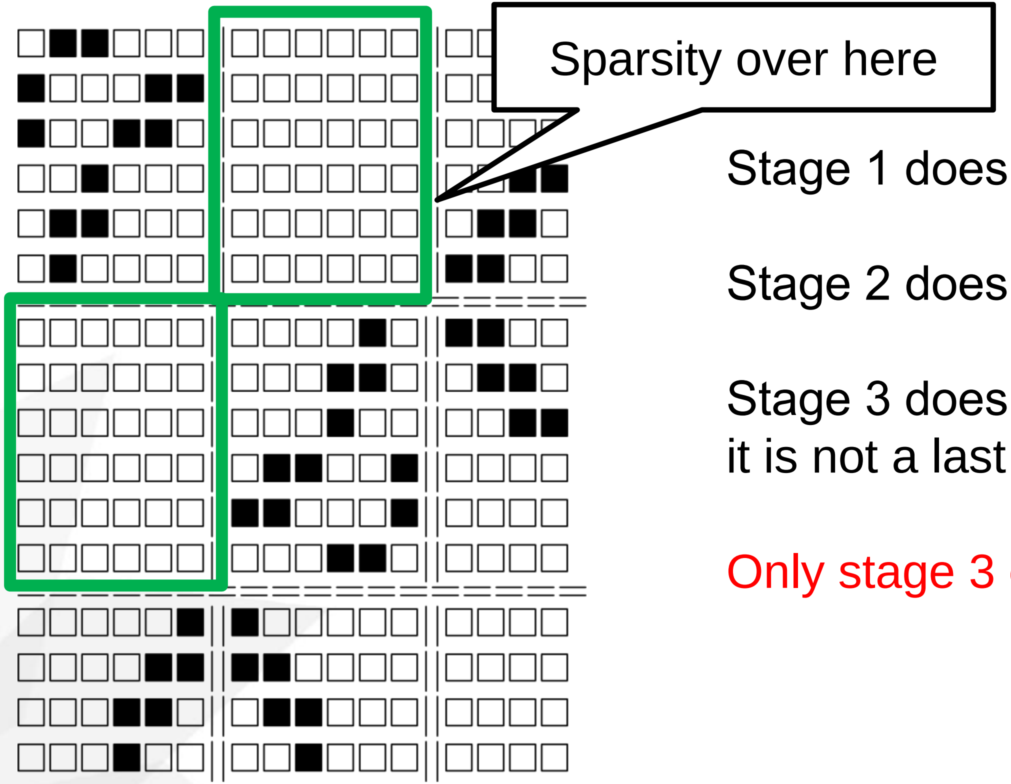
Stage 2 doesn't ruin the sparsity of matrix.

Stage 3 doesn't ruin the sparsity of matrix if it is not a last iteration.

Only stage 3 of last iteration ruins sparsity.

Supernodal Floyd-Warshall

Vertex reordering



Stage 1 doesn’t ruin the sparsity of matrix.

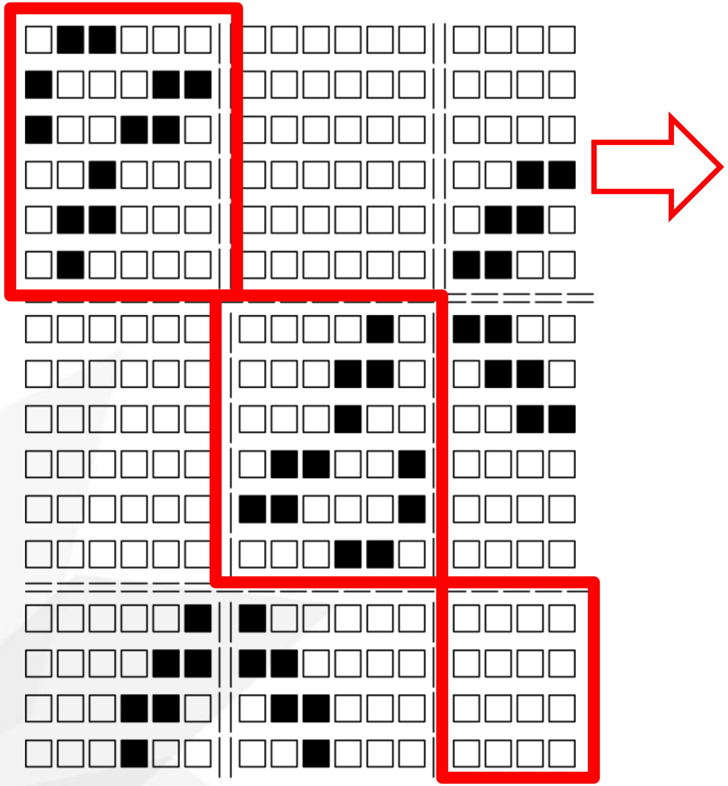
Stage 2 doesn’t ruin the sparsity of matrix.

Stage 3 doesn’t ruin the sparsity of matrix if it is not a last iteration.

Only stage 3 of last iteration ruins sparsity.

Supernodal Floyd-Warshall

Vertex reordering



Stage 1 doesn’t ruin the sparsity of matrix.

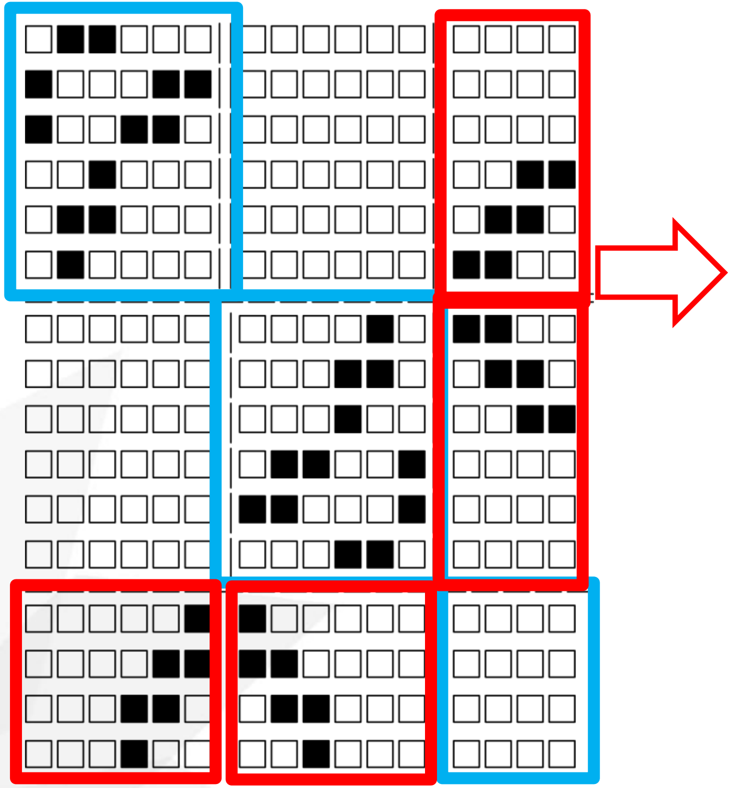
Stage 2 doesn’t ruin the sparsity of matrix.

Stage 3 doesn’t ruin the sparsity of matrix if it is not a last iteration.

Only stage 3 of last iteration ruins sparsity.

Supernodal Floyd-Warshall

Vertex reordering



Stage 1 doesn’t ruin the sparsity of matrix.

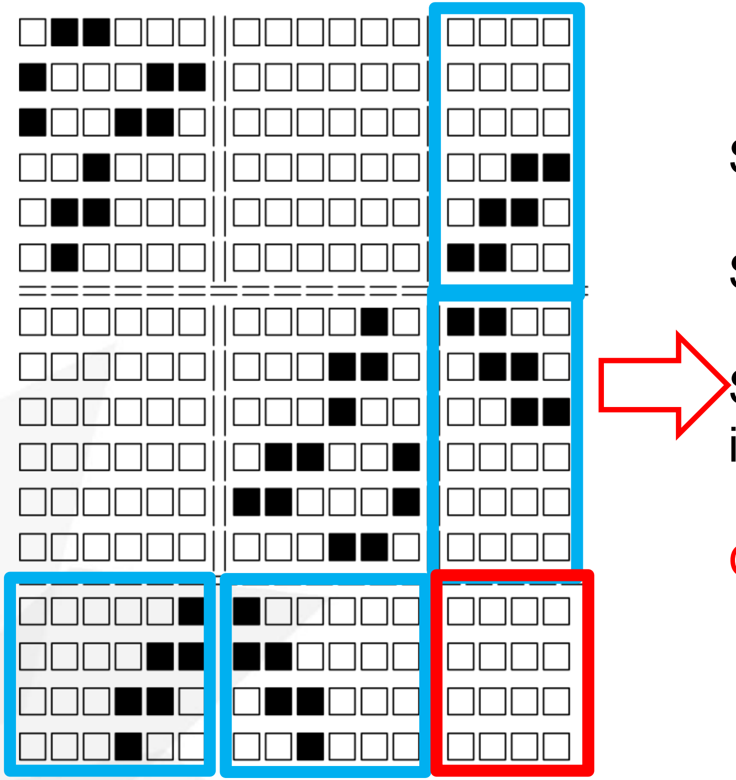
Stage 2 doesn’t ruin the sparsity of matrix.

Stage 3 doesn’t ruin the sparsity of matrix if it is not a last iteration.

Only stage 3 of last iteration ruins sparsity.

Supernodal Floyd-Warshall

Vertex reordering



Stage 1 doesn’t ruin the sparsity of matrix.

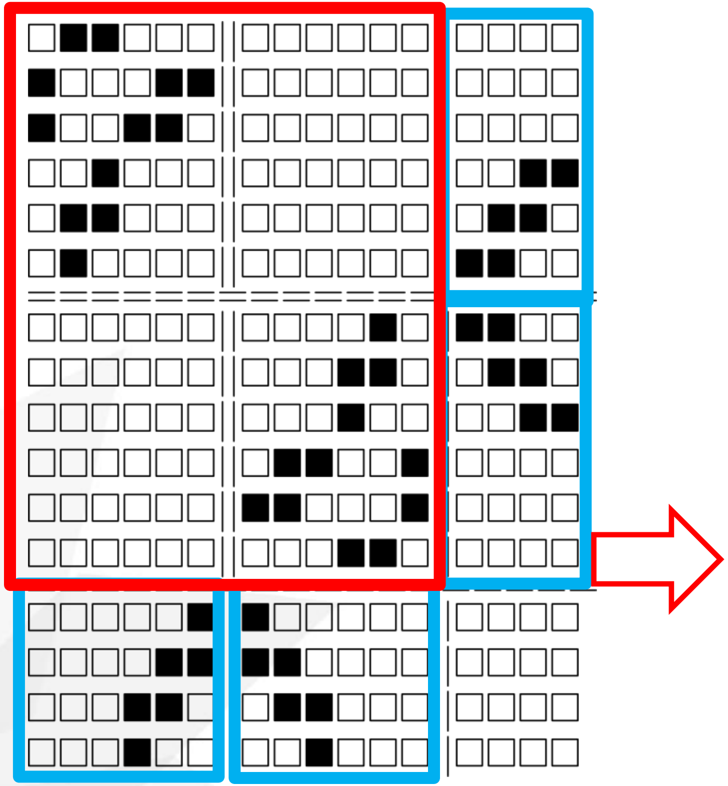
Stage 2 doesn’t ruin the sparsity of matrix.

Stage 3 doesn’t ruin the sparsity of matrix if it is not a last iteration.

Only stage 3 of last iteration ruins sparsity.

Supernodal Floyd-Warshall

Vertex reordering



Stage 1 doesn't ruin the sparsity of matrix.

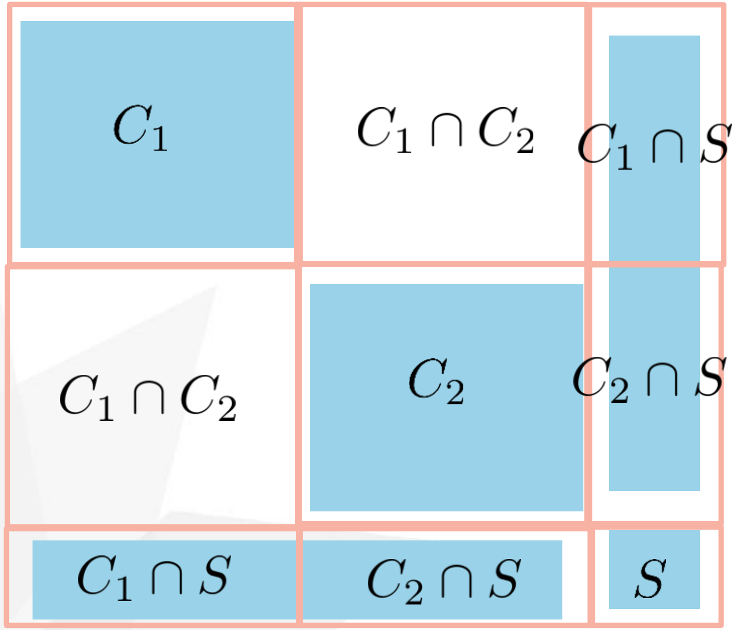
Stage 2 doesn't ruin the sparsity of matrix.

Stage 3 doesn't ruin the sparsity of matrix if it is not a last iteration.

Only stage 3 of last iteration ruins sparsity.

Supernodal Floyd-Warshall

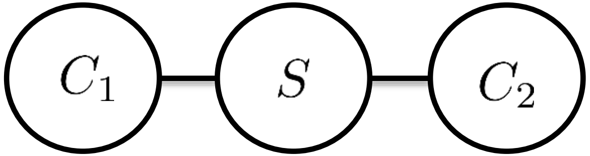
Nested Dissection Ordering



If we can find S, C_1, C_2 satisfies under conditions, it can be reordered nicely.

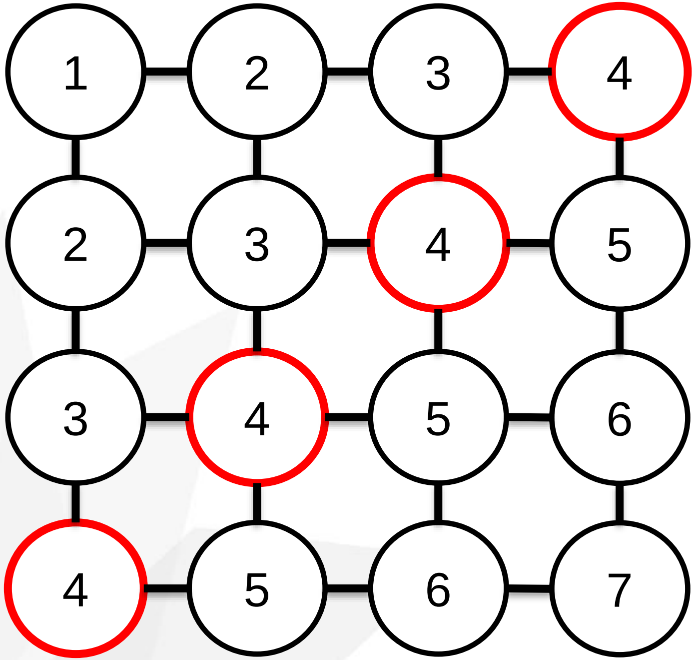
1. $S \cup C_1 \cup C_2 = V$, 2. $C_1 \cap C_2 = \emptyset$
3. $|C_1| \approx |C_2|$,4. $|S| \ll |C_1|, |C_2|$

Nested dissection makes graph to two bisection parts and small separator. Index of separators will be the biggest.



Supernodal Floyd-Warshall

Nested Dissection Ordering



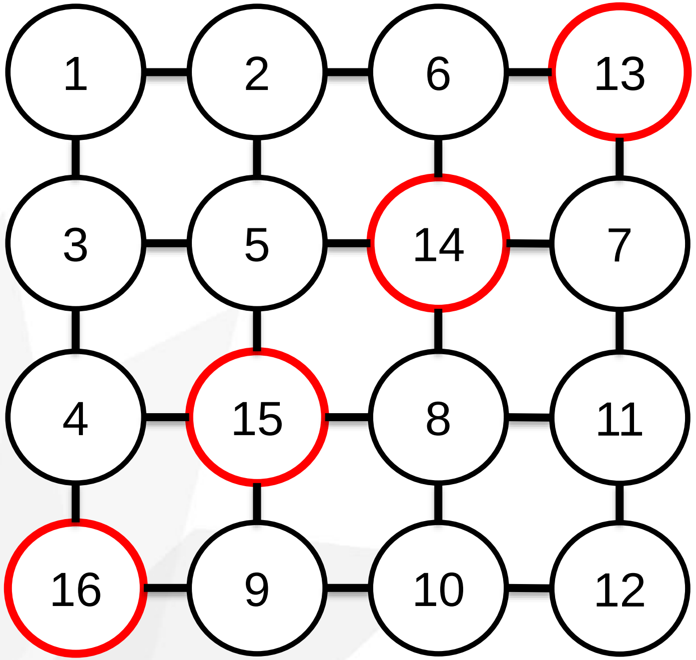
There are no much detail how to implement nested dissection. ☹
However, this kind of problem is popular for a Cholesky factorization.☺
There are some tools support this like Matis.
From the book which referenced at paper,
it can use Cuthill–McKee algorithm to make Nested Dissection order.

Naively saying, it works as bellow.

1. Choose the nodes with minimum outgoing edges.
2. Do BFS to check Depths.
3. Choose the set of medium depth as S.

Supernodal Floyd-Warshall

Nested Dissection Ordering



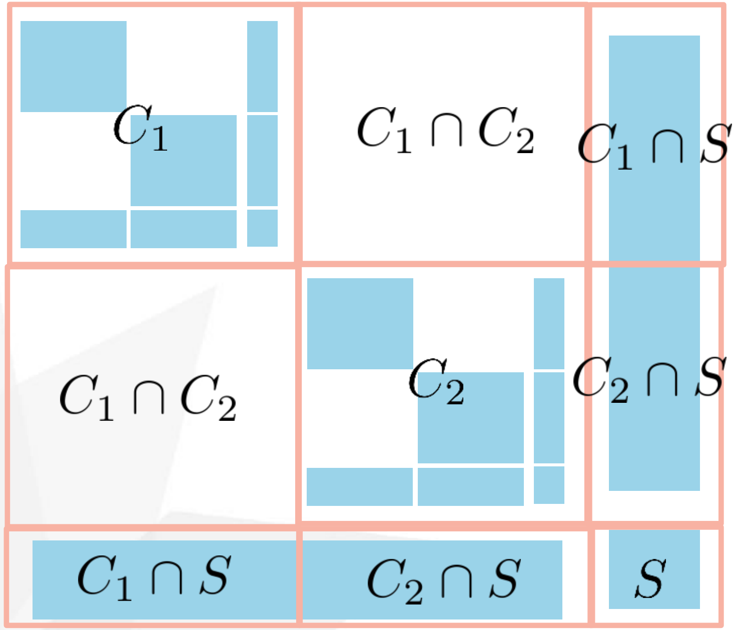
There are no much detail how to implement nested dissection. ☹
However, this kind of problem is popular for a Cholesky factorization.☺
There are some tools support this like Matis.
From the book which referenced at paper, it can use Cuthill–McKee algorithm to make Nested Dissection order.

Naively saying, it works as bellow.

1. Choose the nodes with minimum outgoing edges.
2. Do BFS to check Depths.
3. Choose the set of medium depth as S.

Supernodal Floyd-Warshall

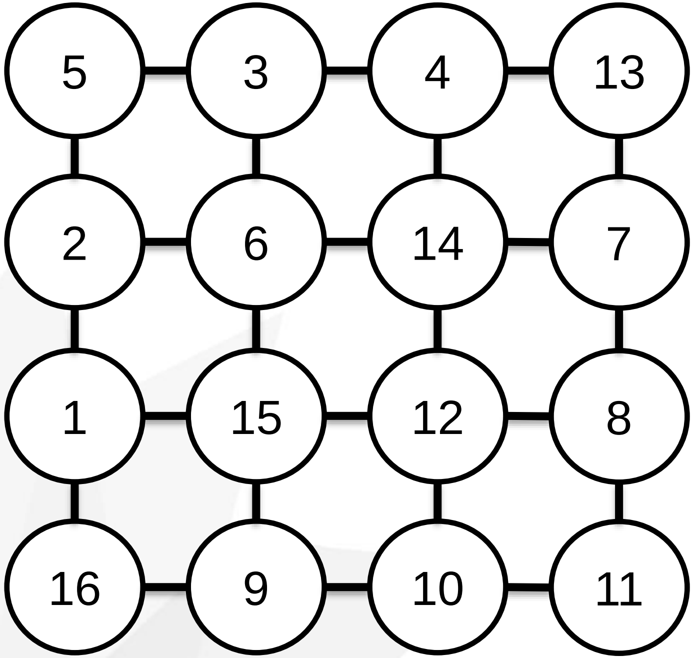
Nested Dissection Ordering

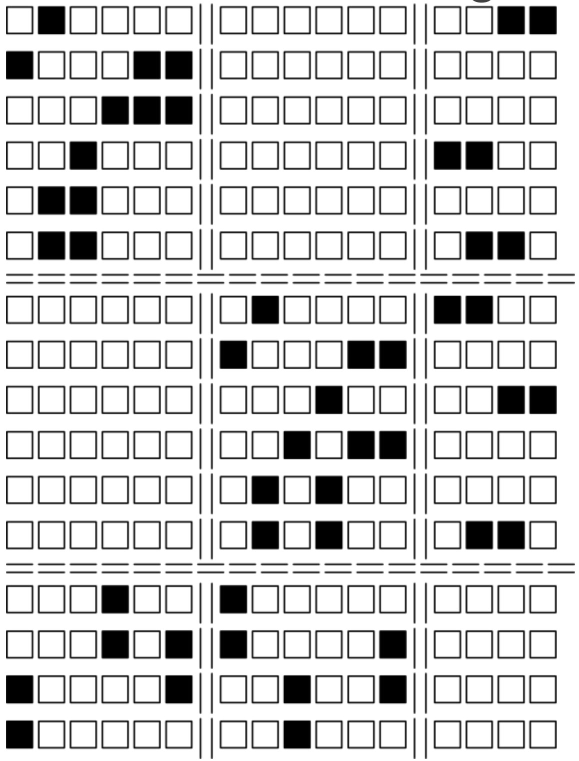


It is called as a nested dissection ordering.
Since, it can be nested recursively in C_1, C_2 .

Supernodal Floyd-Warshall

Nested Dissection Ordering



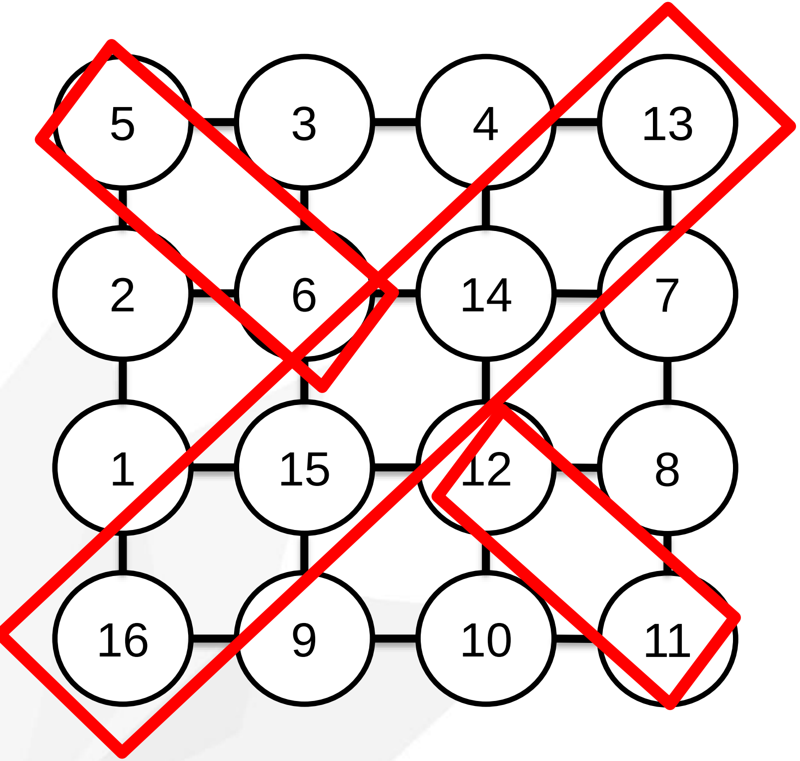


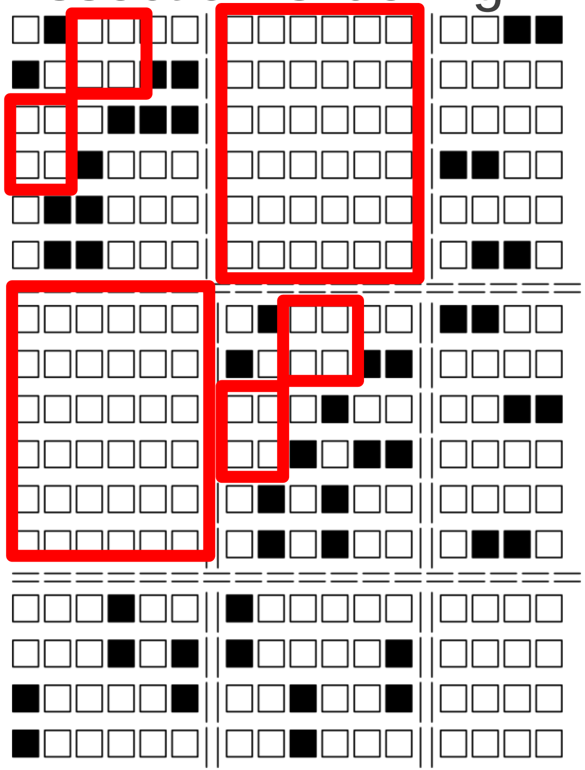
If using a nested dissection recursively, results will be left matrix. Which has more sparsity in it.

Notice that separator will have the biggest index.

Supernodal Floyd-Warshall

Nested Dissection Ordering



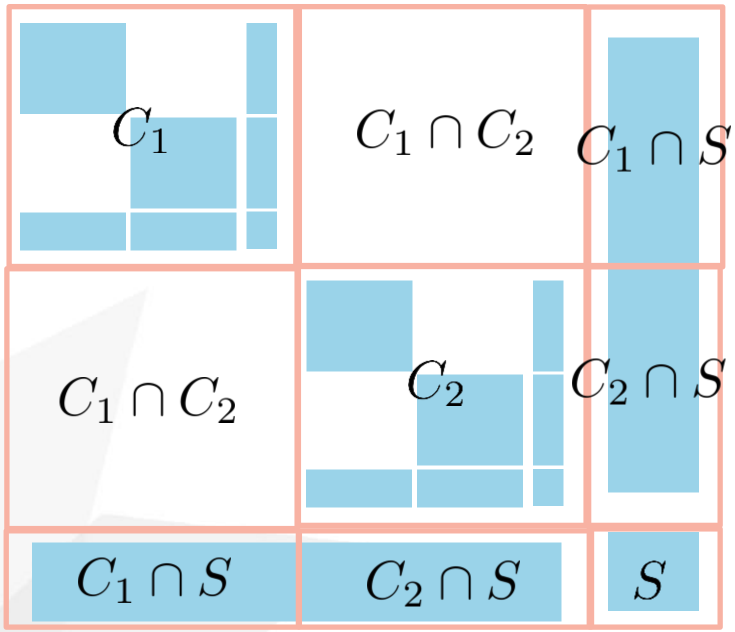


If using a nested dissection recursively, results will be left matrix. Which has more sparsity in it.

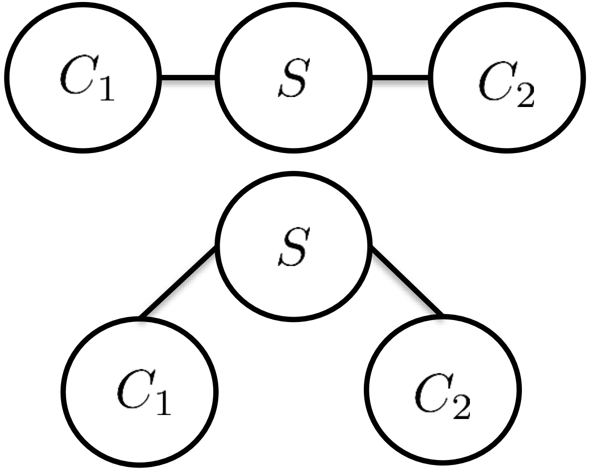
Notice that separator will have the biggest index.

Supernodal Floyd-Warshall

Elimination tree and supernodes

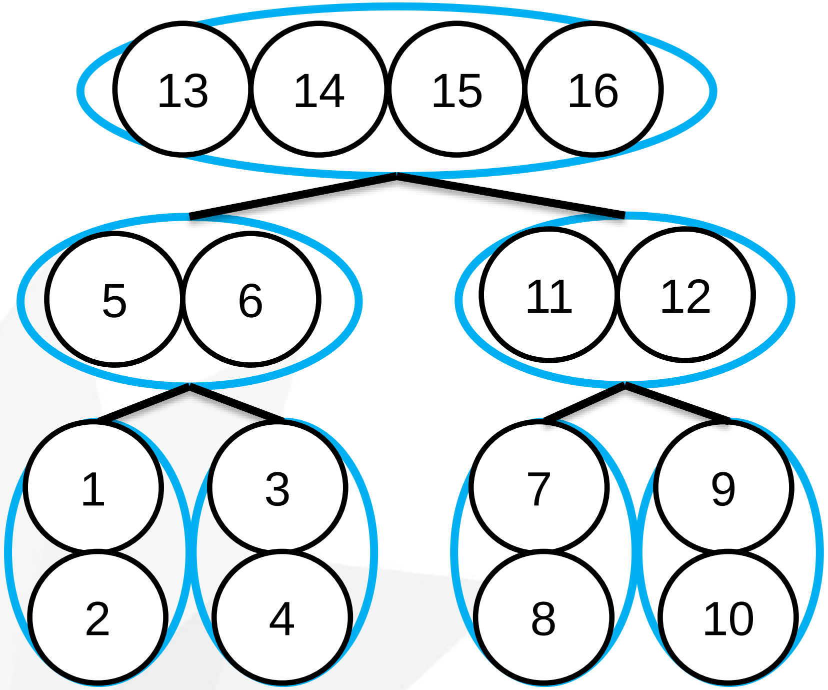


Nested dissection can easily change graph to tree.
It called elimination tree.
Sibilings have no connection between.



Supernodal Floyd-Warshall

Elimination tree and supernodes

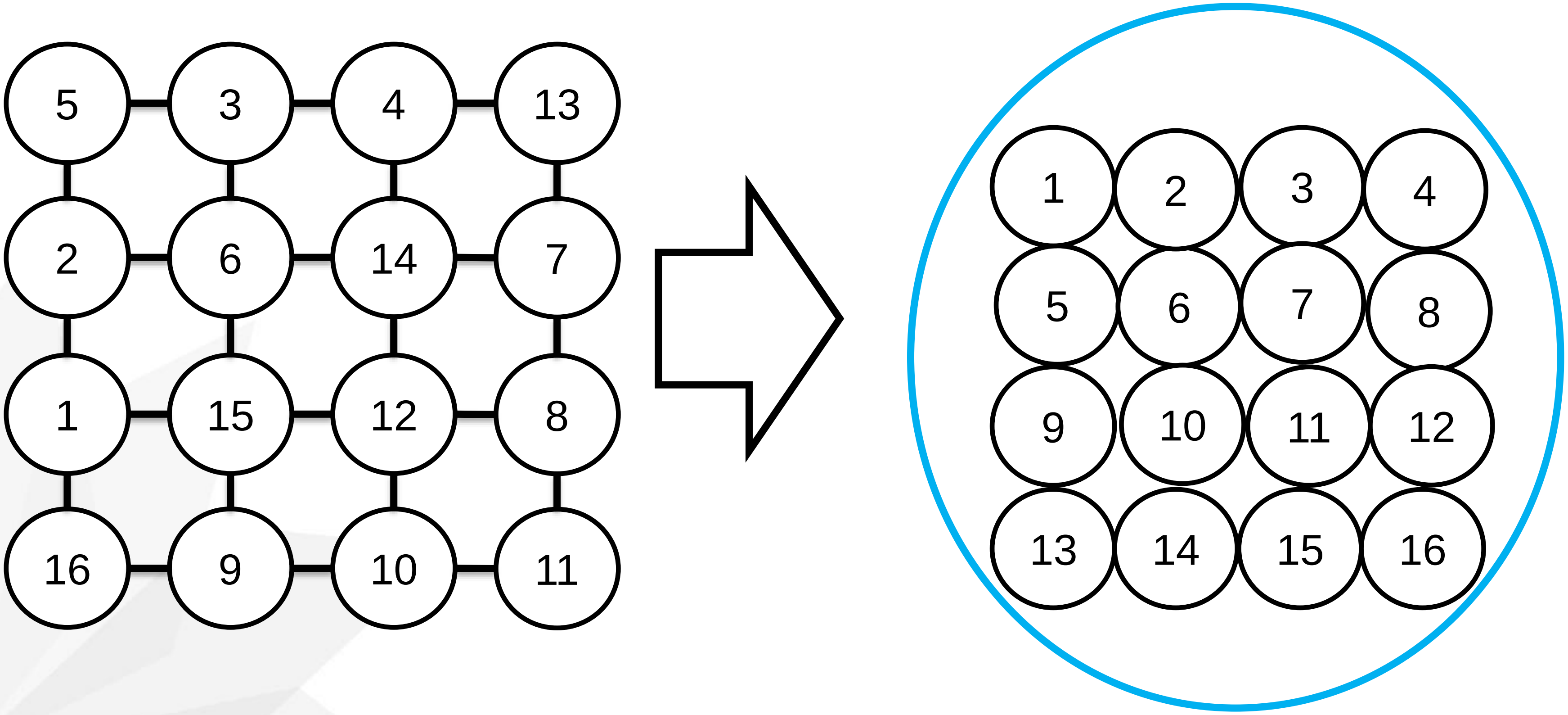


If it changes to an elimination tree.
There is some nodes that containing many vertices in it.

Now vertices has covered in a super nodes.
Calculation will be done over a super nodes.
Detail example will be follows.

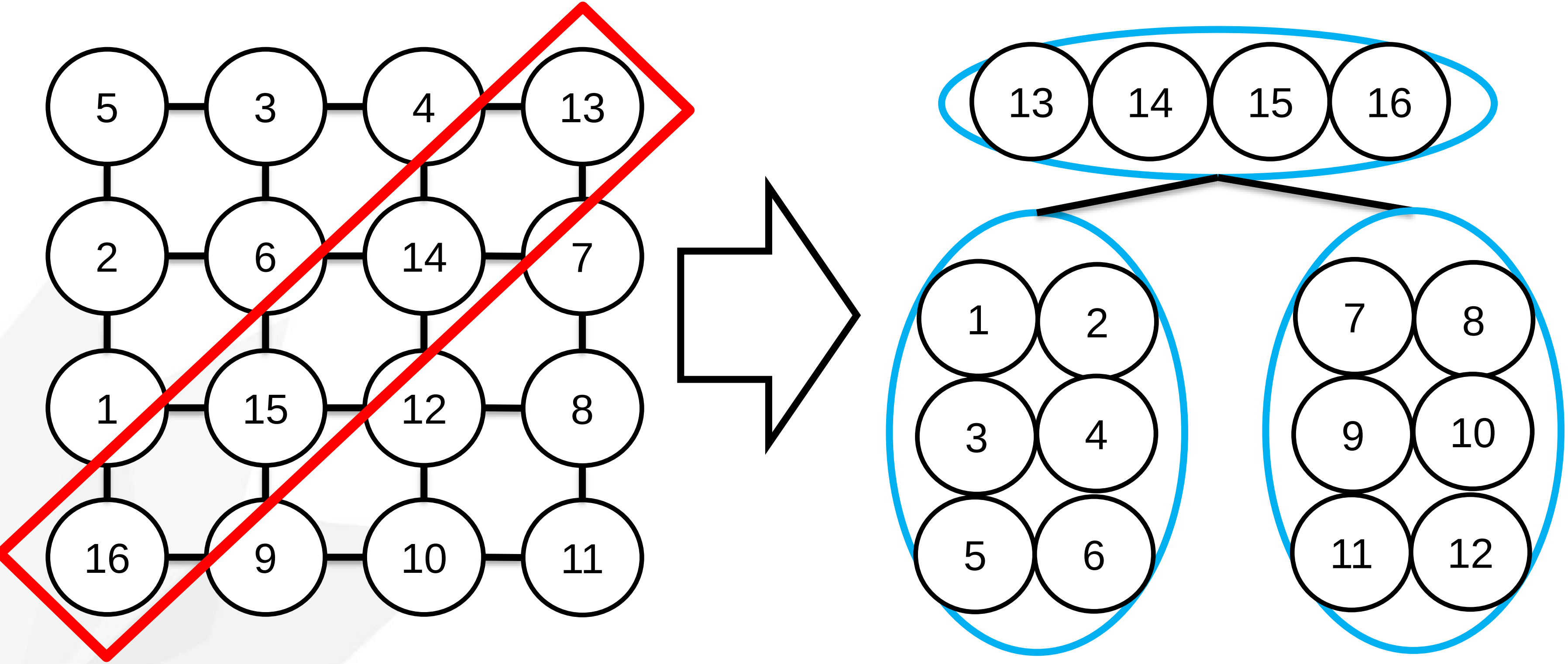
Supernodal Floyd-Warshall

Elimination tree and supernodes



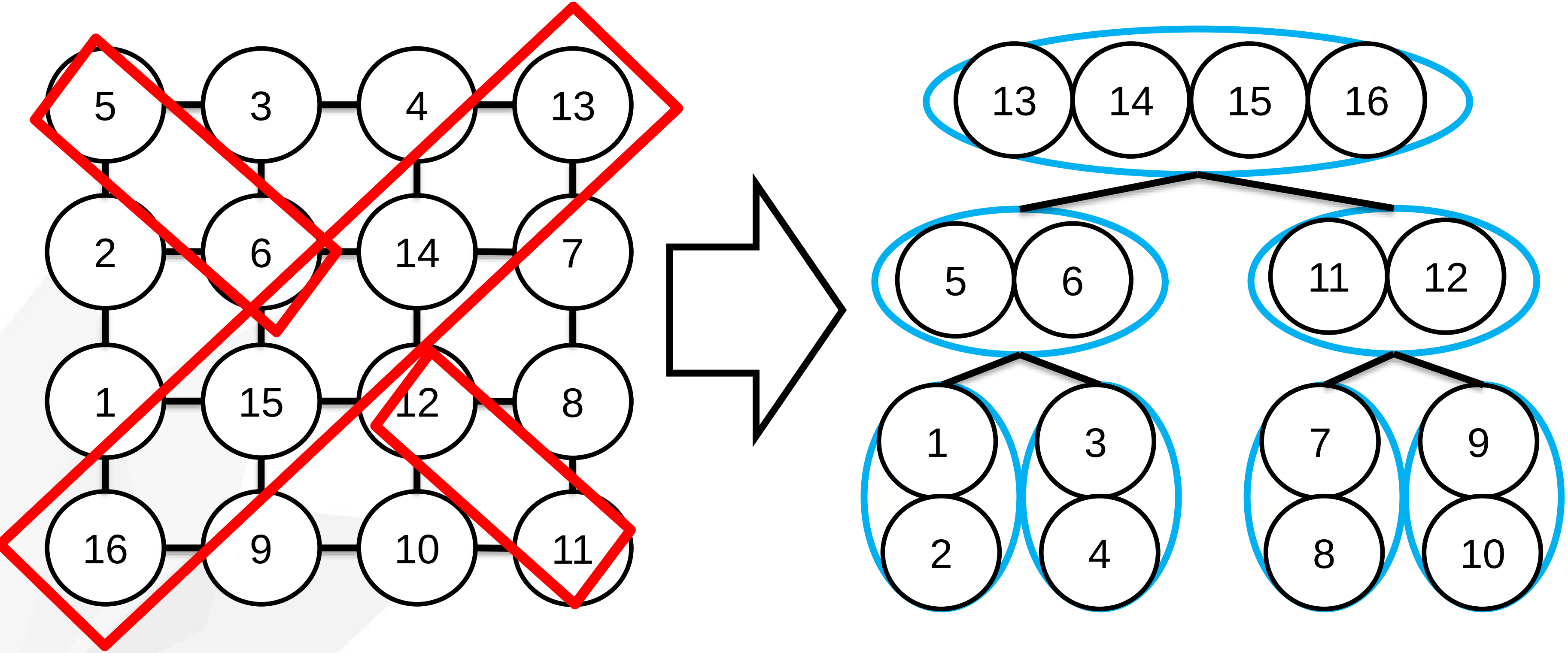
Supernodal Floyd-Warshall

Elimination tree and supernodes



Supernodal Floyd-Warshall

Elimination tree and supernodes



Supernodal Floyd-Warshall

Sequential SuperFW algorithm

Algorithm 3 The SuperFw algorithm

n_s := Number of supernodes
function SuperFw($G=(V,E)$):
  for $k=\{1, 2\ldots,n_s\}$ **do**:
  Diagonal Update
    $A(k,k) \leftarrow$ Floyd-Warshall$(A(k,k))$
  Panel Update
    for $i \in \boxed{\mathcal{A}(k)\cup\mathcal{D}(k)}$ **do**
      $A(i,k) \leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k)$
      $A(k,i) \leftarrow A(k,i) \oplus A(k,k) \otimes A(k,i)$
  MinPlus Outer Product
    for $(i,j) \in \boxed{\{\mathcal{A}(k)\cup\mathcal{D}(k)\}\times\{\mathcal{A}(k)\cup\mathcal{D}(k)\}}$ **do**:
      $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$

Figure 16. The supernodal Floyd-Warshall algorithm

A Supernodal Floyd-Warshall algorithm is a quietly same with Blocked Floyd-Warshall.
Major calculation reduction come from the elimination tree.
It checks only ancestor and descendants.

Supernodal Floyd-Warshall

Sequential SuperFW algorithm

Algorithm 2 A blocked version of FLOYD-WAR
rithm for APSP

function BLOCKEDFLOYDWARSHALL(A):
    for $k = \{1, 2 \ldots, n_b\}$ **do**:
Diagonal Update
        $A(k,k) \leftarrow$ FLOYD-WARSHALL$(A(k,k))$
Panel Update
        $A(k,:) \leftarrow A(k,:) \bigoplus A(k,k) \otimes A(k,:)$
        $A(:,k) \leftarrow A(:,k) \bigoplus A(:,k) \otimes A(k,k)$
MinPlus Outer Product
        for $i = \{1, 2 \ldots, n_b\}, i \neq k$ **do**:
            for $j = \{1, 2 \ldots, n_b\}, j \neq k$ **do**:
                $A(i,j) \leftarrow A(i,j) \bigoplus A(i,k) \otimes A(k,j)$
    Return A

Algorithm 3 The SUPERFW algorithm

n_s := Number of supernodes
function SUPERFW($G = (V, E)$):
    for $k = \{1, 2 \ldots, n_s\}$ **do**:
Diagonal Update
        $A(k,k) \leftarrow$ FLOYD-WARSHALL$(A(k,k))$
Panel Update
        for $i \in \mathcal{A}(k) \cup \mathcal{D}(k)$ **do**
            $A(i,k) \leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k)$
            $A(k,i) \leftarrow A(k,i) \oplus A(k,k) \otimes A(k,i)$
MinPlus Outer Product
        for $(i,j) \in \{\mathcal{A}(k) \cup \mathcal{D}(k)\} \times \{\mathcal{A}(k) \cup \mathcal{D}(k)\}$ **do**:
            $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$

Supernodal Floyd-Warshall

Sequential SuperFW algorithm

Algorithm 2 A blocked version of FLOYD-WAR rithm for APSP

```
function BLOCKEDFLOYDWARSHALL(A):
    for k = {1, 2…,n_b} do:
    Diagonal Update
        A(k,k) ← FLO
    Panel Update
        A(k,:) ← A(k,:) ⊕ A(k,k)⊗A(k,:)
        A(:,k) ← A(:,k) ⊕ A(:,k)⊗A(k,k)
    MinPlus Outer Product
        for i = {1, 2…,n_b}, i ≠ k do:
            for j = {1, 2…,n_b}, j ≠ k do:
                A(i,j) ← A(i,j) ⊕ A(i,k)⊗A(k,j)
    Return A
```

Algorithm 3 The SUPERFW algorithm

```
n_s := Number of supernodes
function SUPERFW(G=(V,E)):
    for k = {1, 2…,n_s} do:
        HALL(A(k,k))
    Panel Update
        for i ∈ 𝒜(k)∪𝒟(k) do
            A(i,k) ← A(i,k) ⊕ A(i,k)⊗A(k,k)
            A(k,i) ← A(k,i) ⊕ A(k,k)⊗A(k,i)
    MinPlus Outer Product
        for (i,j) ∈ {𝒜(k)∪𝒟(k)}×{𝒜(k)∪𝒟(k)} do:
            A(i,j) ← A(i,j) ⊕ A(i,k)⊗A(k,j)
```

Compute only over ancestors and descendant

Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Algorithm 3 The SuperFw algorithm

```
n_s := Number of supernodes
function SuperFw(G=(V,E)):
    for k={1, 2...,n_s} do:
  Diagonal Update
        A(k,k) ← Floyd-Warshall(A(k,k))
  Panel Update
        for i ∈ [A(k)∪D(k)] do
            A(i,k) ← A(i,k)⊕ A(i,k)⊗A(k,k)
            A(k,i) ← A(k,i)⊕ A(k,k)⊗A(k,i)
  MinPlus Outer Product
        for (i,j) ∈ [{A(k)∪D(k)}×{A(k)∪D(k)}] do:
            A(i,j) ← A(i,j)⊕ A(i,k)⊗A(k,j)
```

$$n_s := \text{Number of supernodes}$$

Panel Update: for $i \in \boxed{\mathcal{A}(k)\cup\mathcal{D}(k)}$ do

MinPlus Outer Product: for $(i,j) \in \boxed{\{\mathcal{A}(k)\cup\mathcal{D}(k)\}\times\{\mathcal{A}(k)\cup\mathcal{D}(k)\}}$ do:

Let's think about the super nodes at same depth.
Then, ancestors may be same.
However, descendants can't be same.
Here more parallelism points are.

Figure 16. The supernodal Floyd-Warshall algorithm

Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Algorithm 3 The SuperFw algorithm

n_s := Number of supernodes
function SuperFw($G=(V,E)$):
  for $k=\{1, 2 \dots, n_s\}$ **do**:
Diagonal Update
    $A(k,k) \leftarrow$ Floyd-Warshall$(A(k,k))$
Panel Update
    for $i \in \boxed{\mathcal{A}(k) \cup \mathcal{D}(k)}$ **do**
      $A(i,k) \leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k)$
      $A(k,i) \leftarrow A(k,i) \oplus A(k,k) \otimes A(k,i)$
MinPlus Outer Product
    for $(i,j) \in \boxed{\{\mathcal{A}(k) \cup \mathcal{D}(k)\} \times \{\mathcal{A}(k) \cup \mathcal{D}(k)\}}$ **do**:
      $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$

If it calculating over descendants,
it can get a parallelism.
Detail will be following.

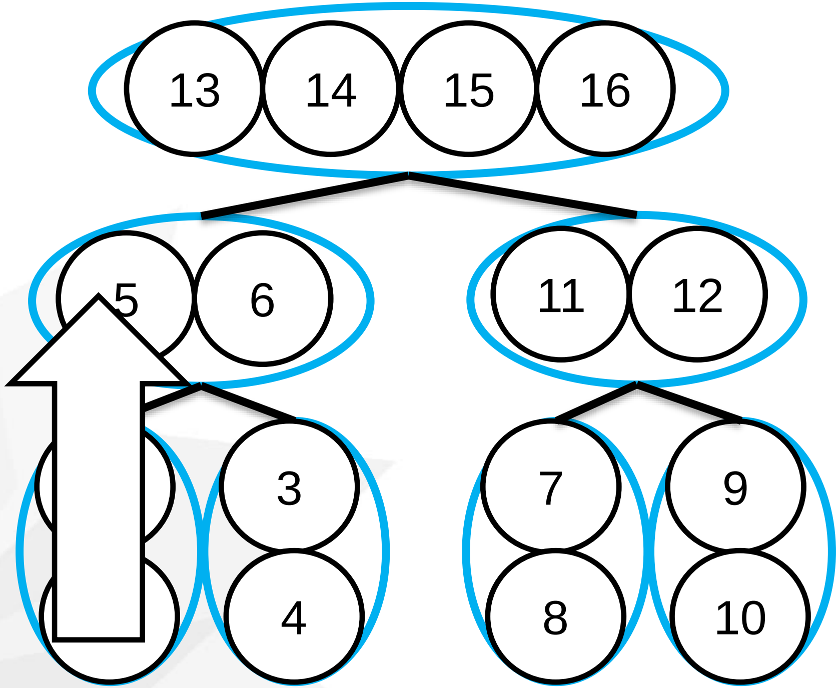
$\mathcal{D}(k) \times \mathcal{D}(k)$
$\mathcal{A}(k) \times \mathcal{D}(k)$
$\mathcal{D}(k) \times \mathcal{A}(k)$
$\mathcal{A}(k) \times \mathcal{A}(k)$

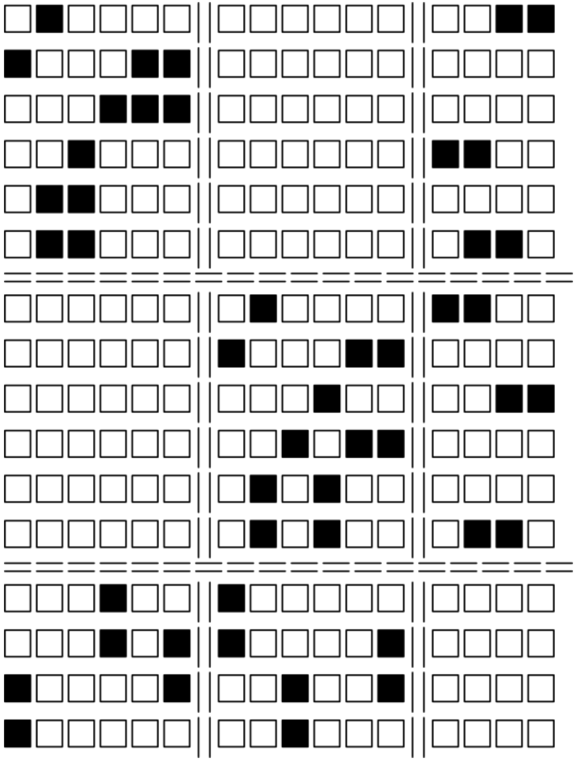
Figure 16. The supernodal Floyd-Warshall algorithm

Supernodal Floyd-Warshall

Parallel SuperFW algorithm

In this paper, it uses a bottom-up approach with a tree reduction.

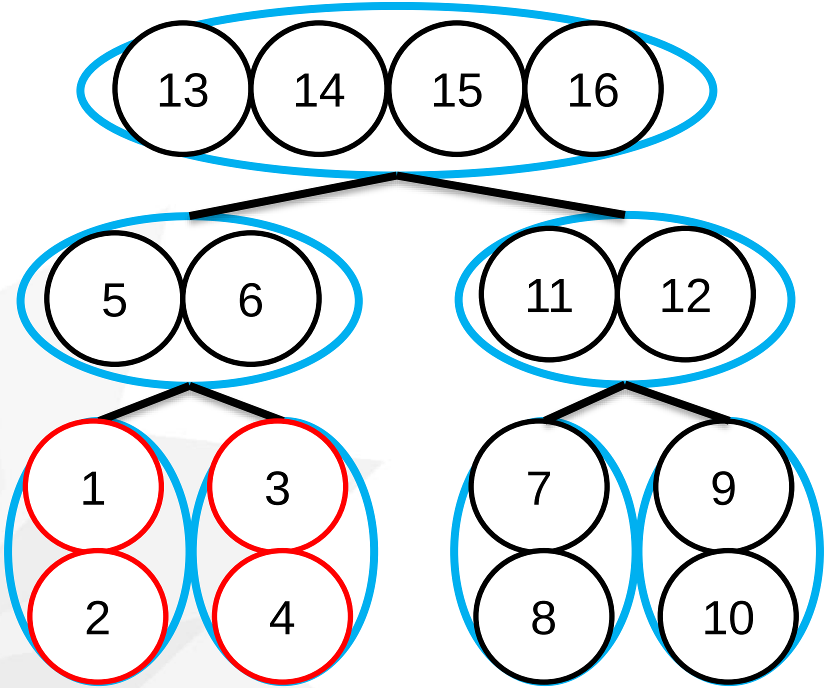


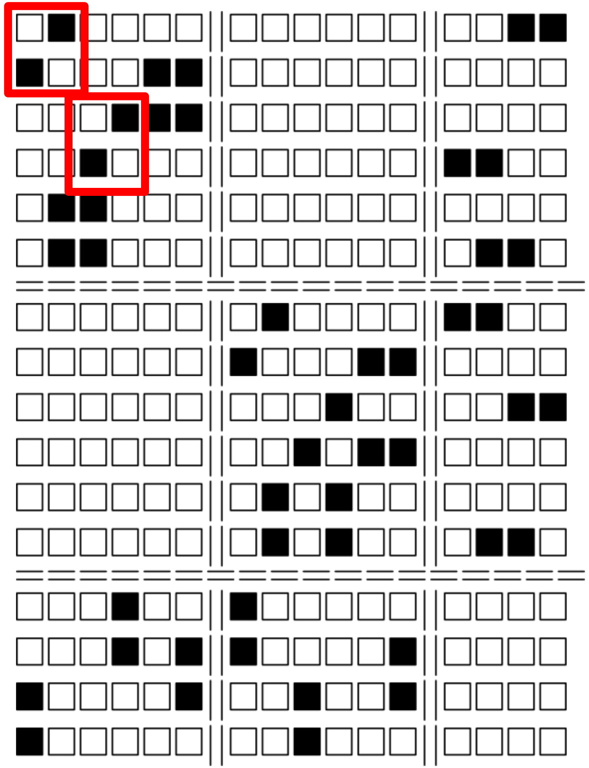


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Start from nodes with lowest height.
Stage 1 is straight forward.

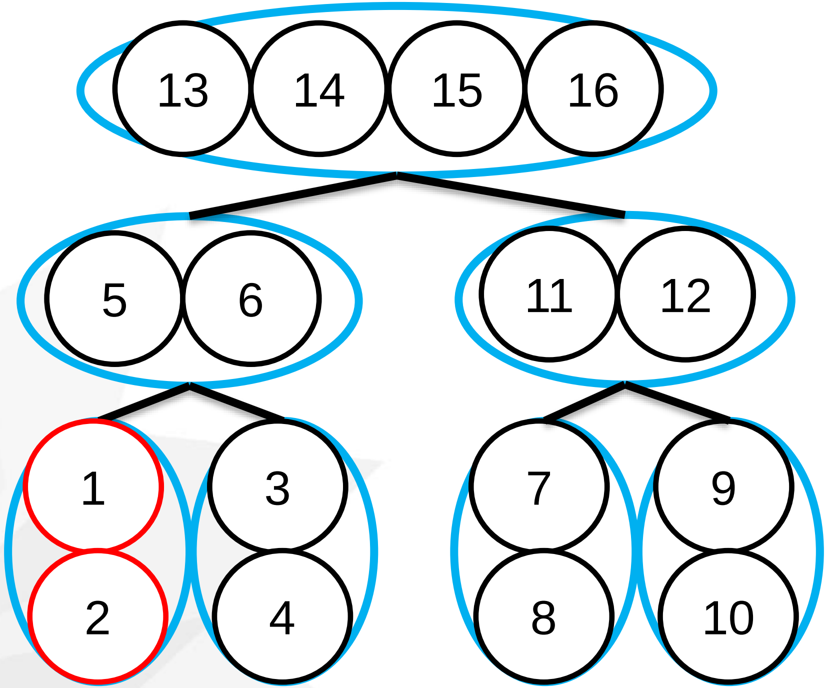


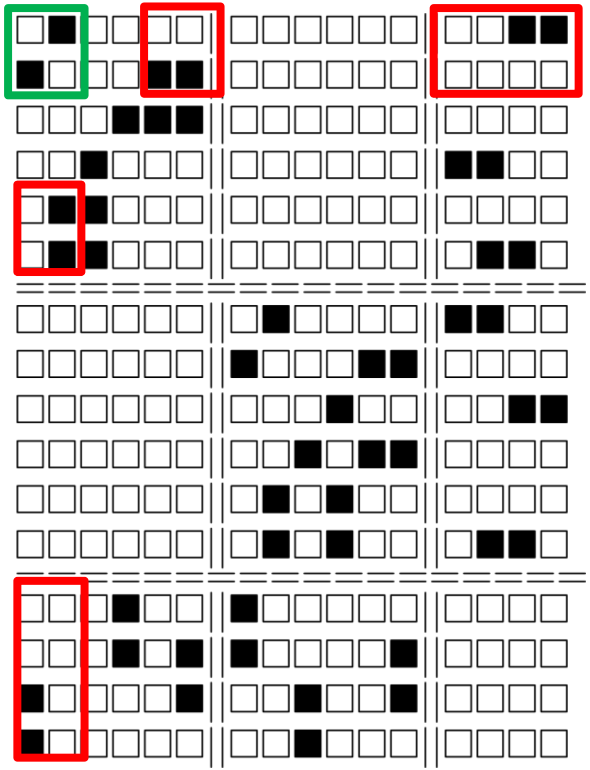


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 2 is straight forward.

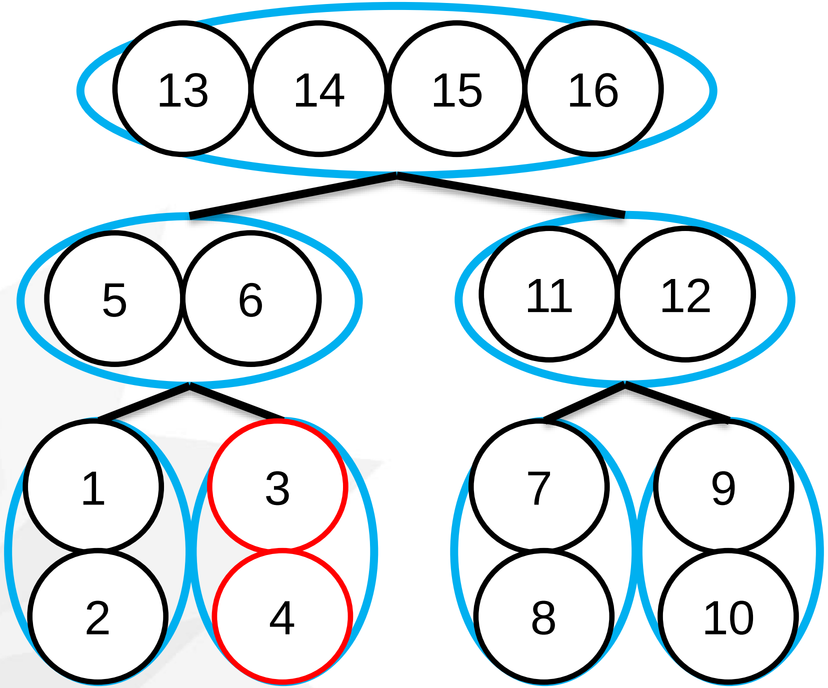


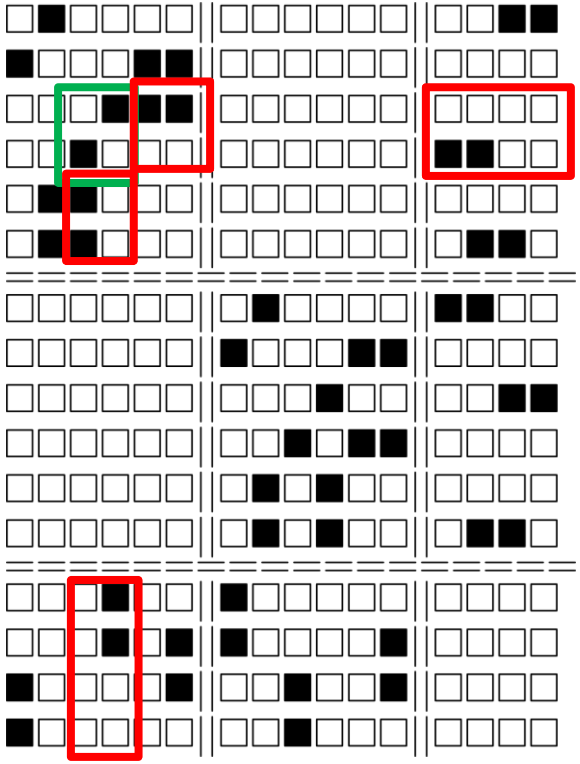


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 2 is straight forward.

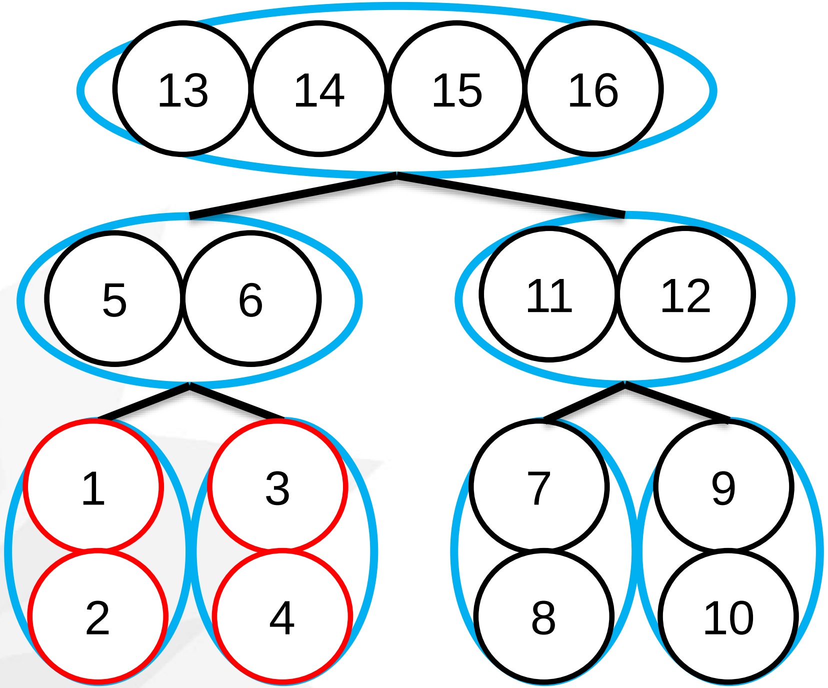


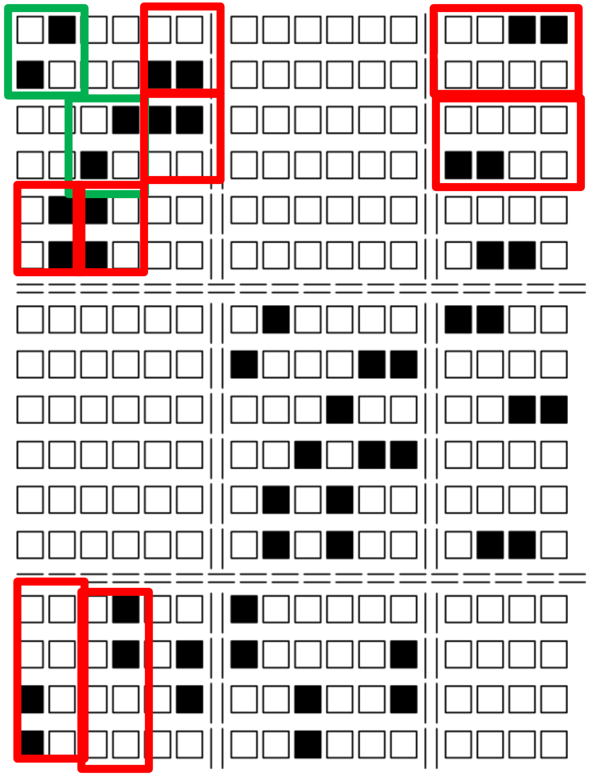


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 2 is straight forward.

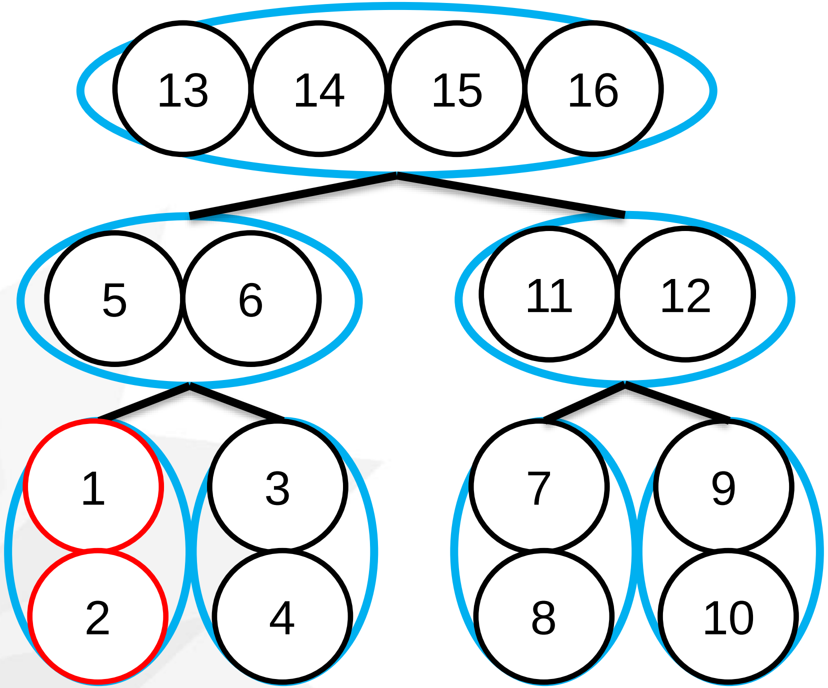


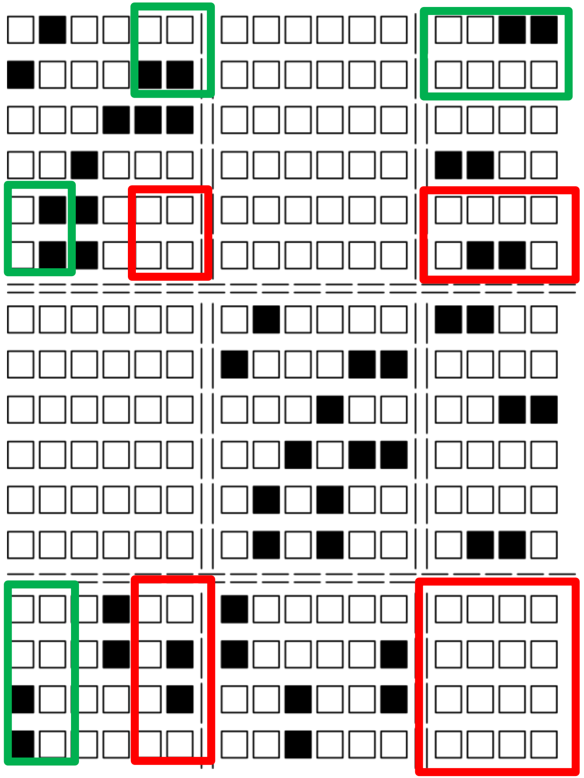


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 3 has collisions.
It resolves this via tree reduction.

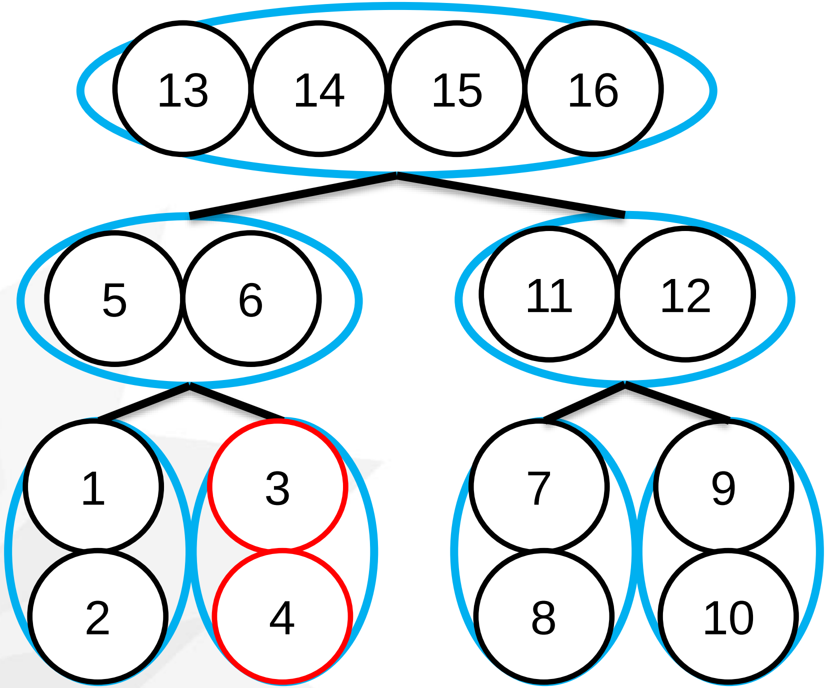


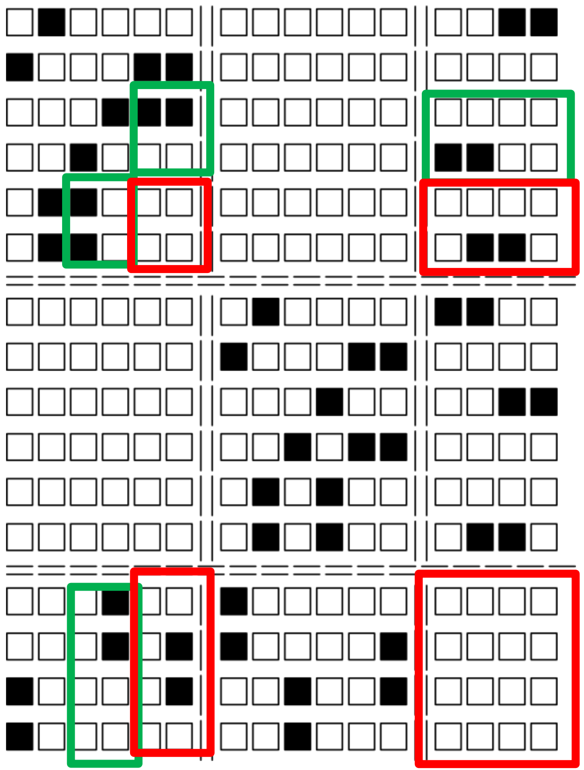


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 3 has collisions.
It resolves this via tree reduction.

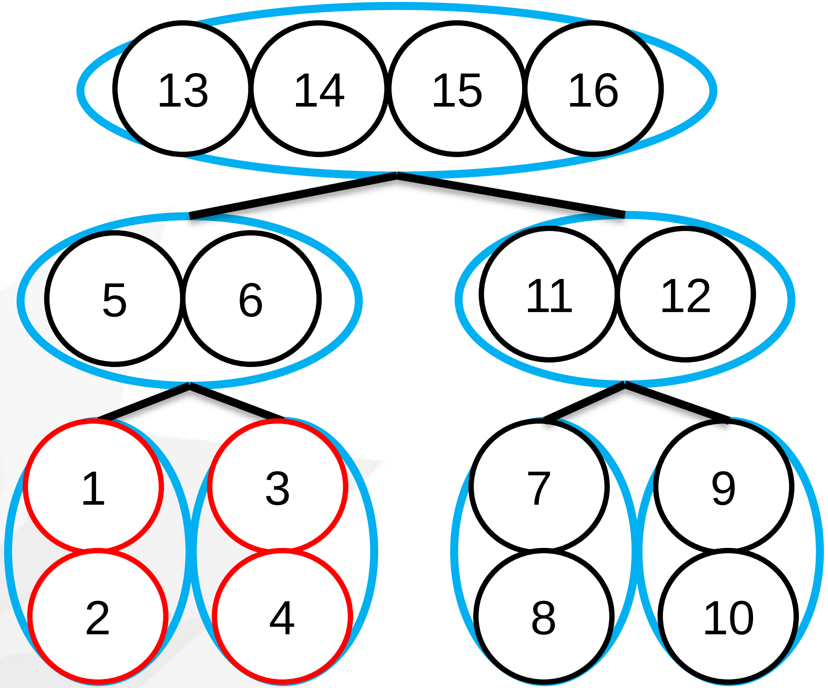


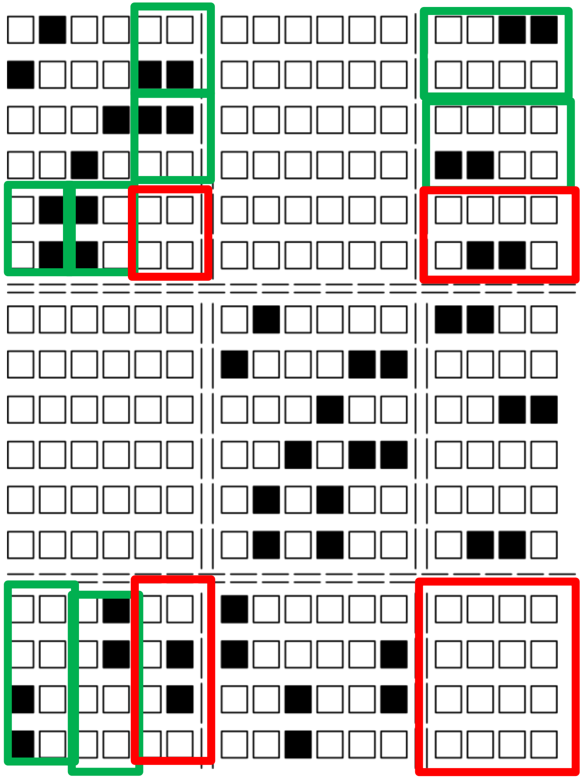


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 3 has collisions.
It resolves this via tree reduction.

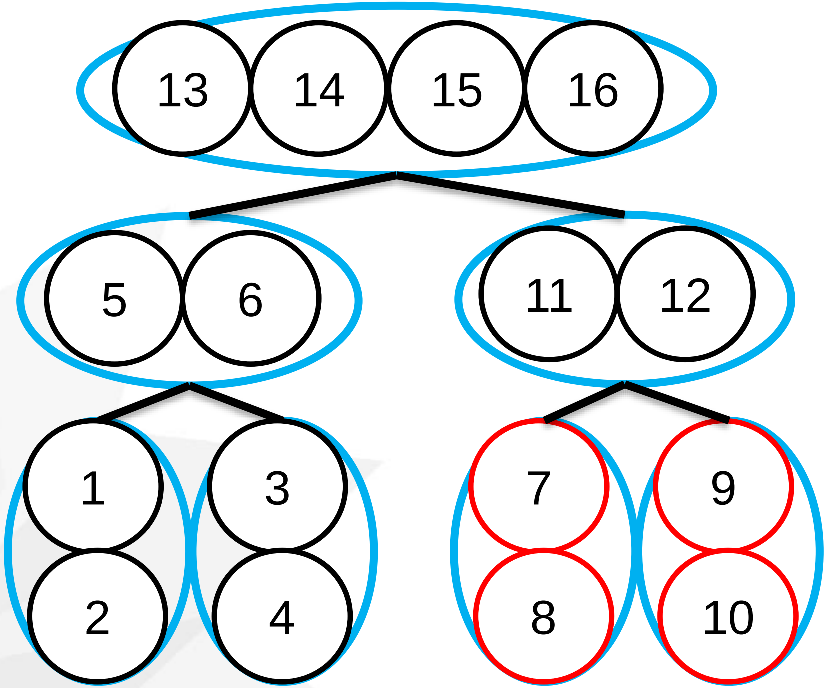


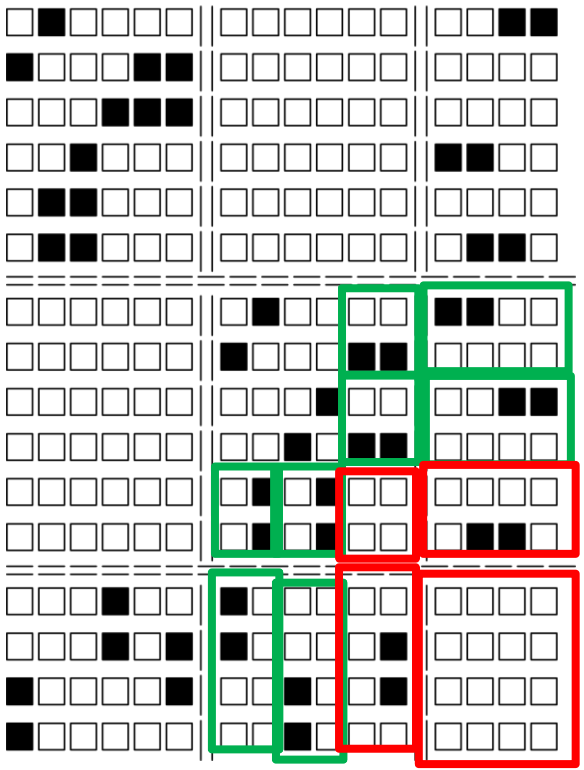


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 3 has collisions.
It resolves this via tree reduction.



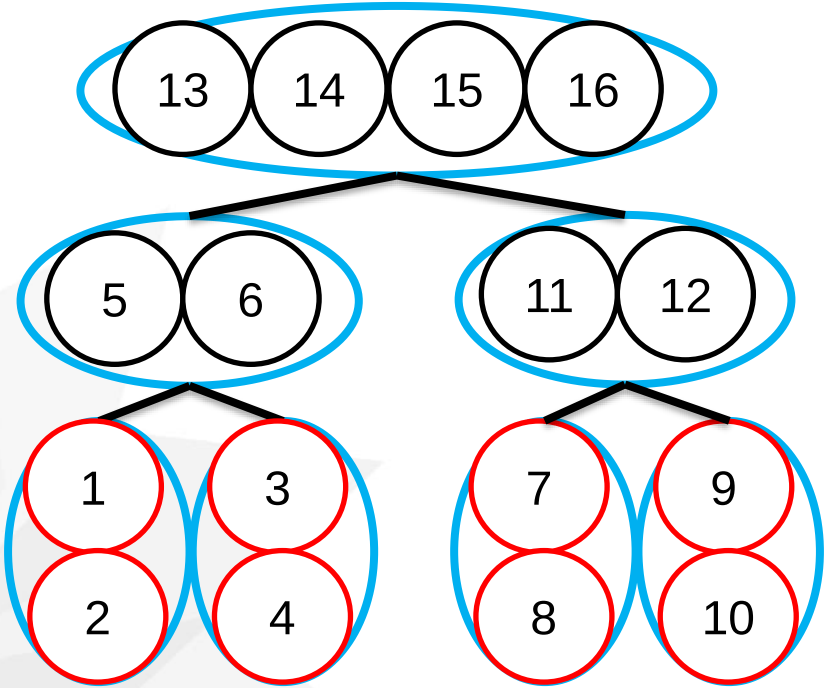


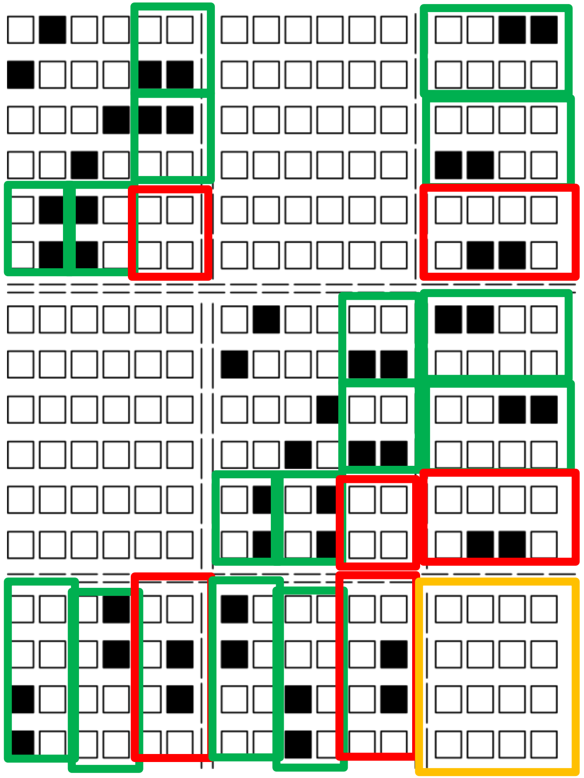
Supernodal Floyd-Warshall

Parallel SuperFW algorithm

Stage 3 has collisions.

It resolves this via tree reduction.

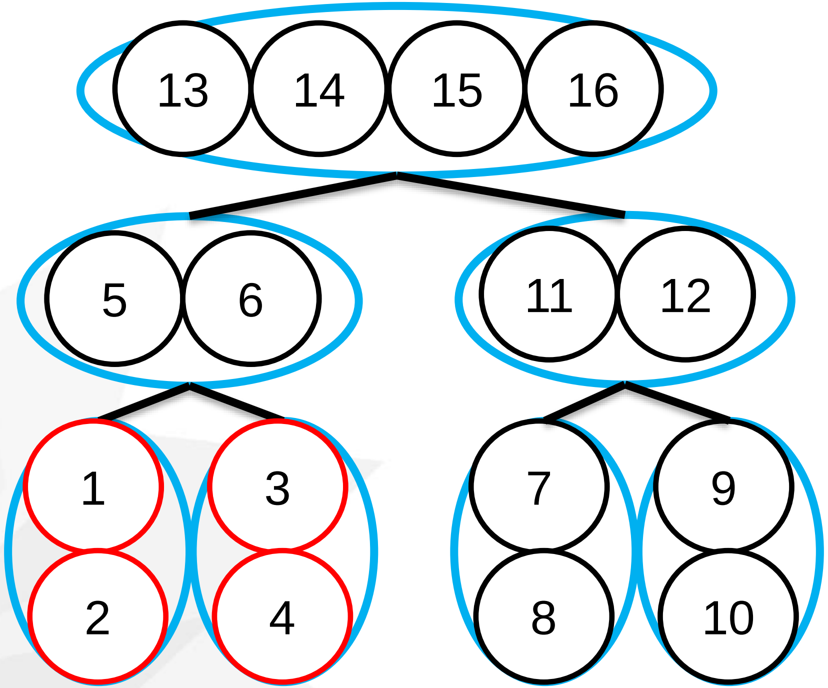


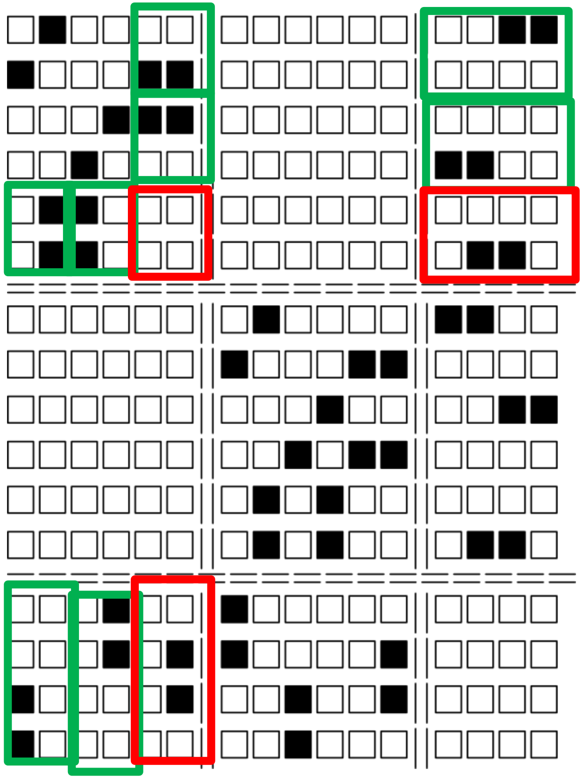


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

These merge between (1,2) and (3,4).

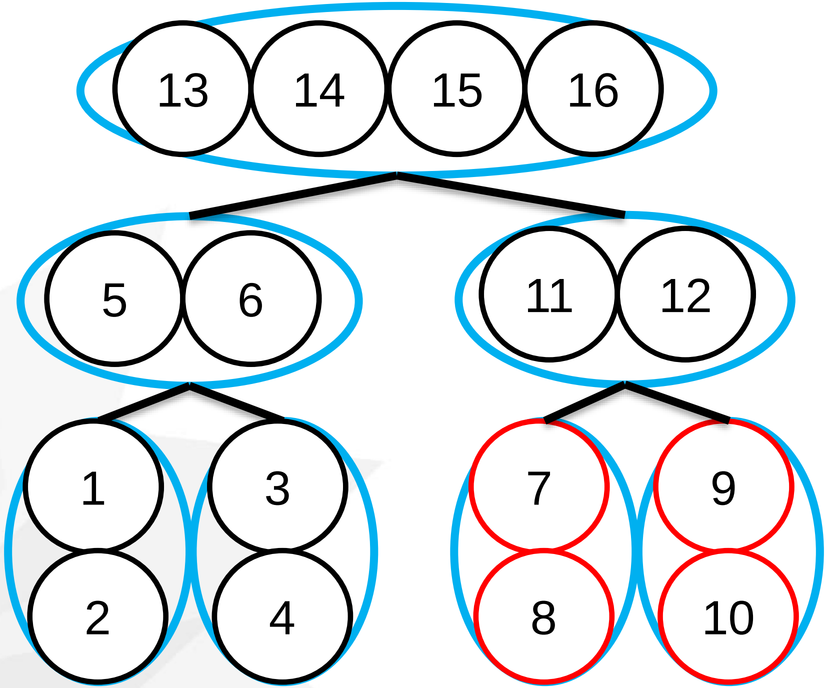


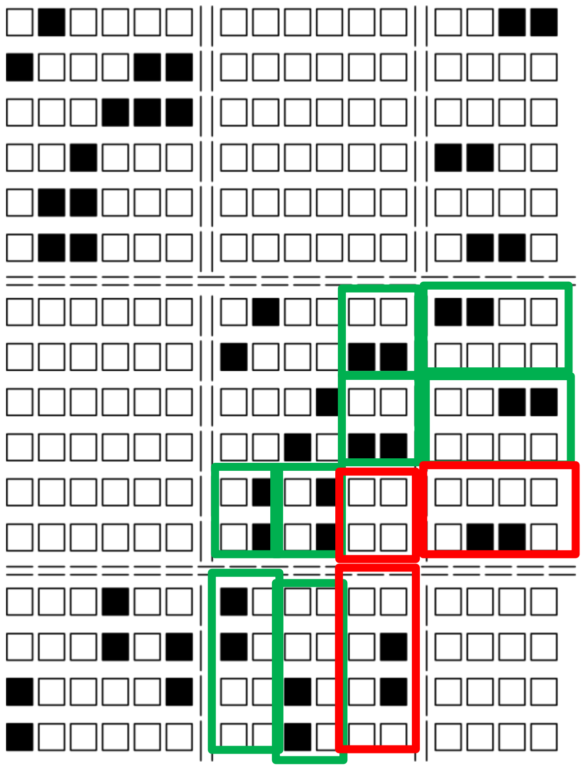


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

These merge between (7,8) and (9,10).

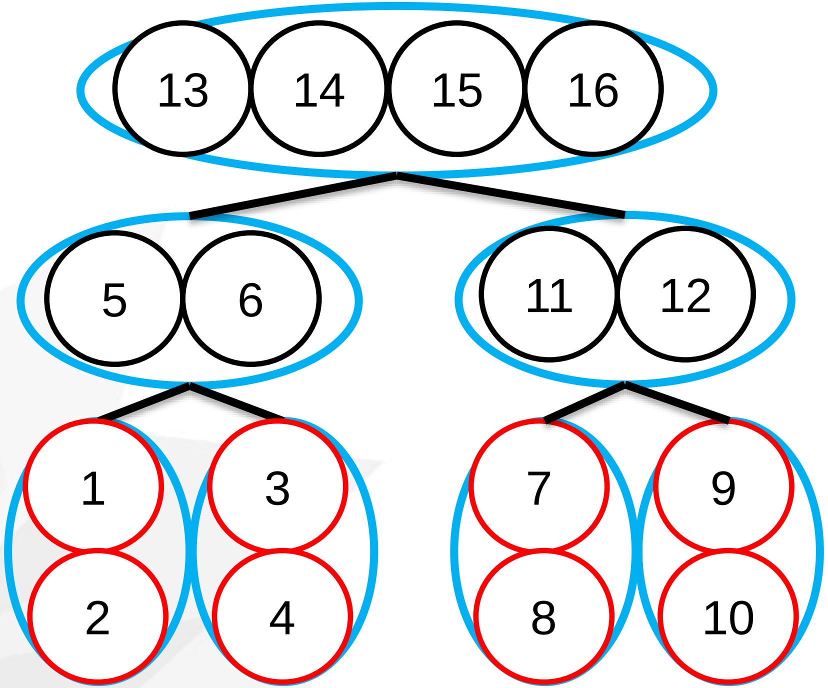


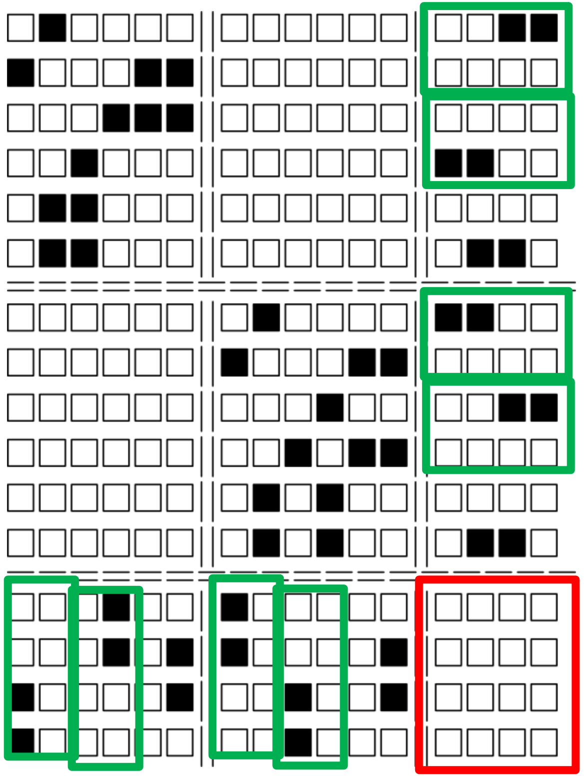


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

These merge among (1,2) ,(3,4), (7,8) and (9,10).

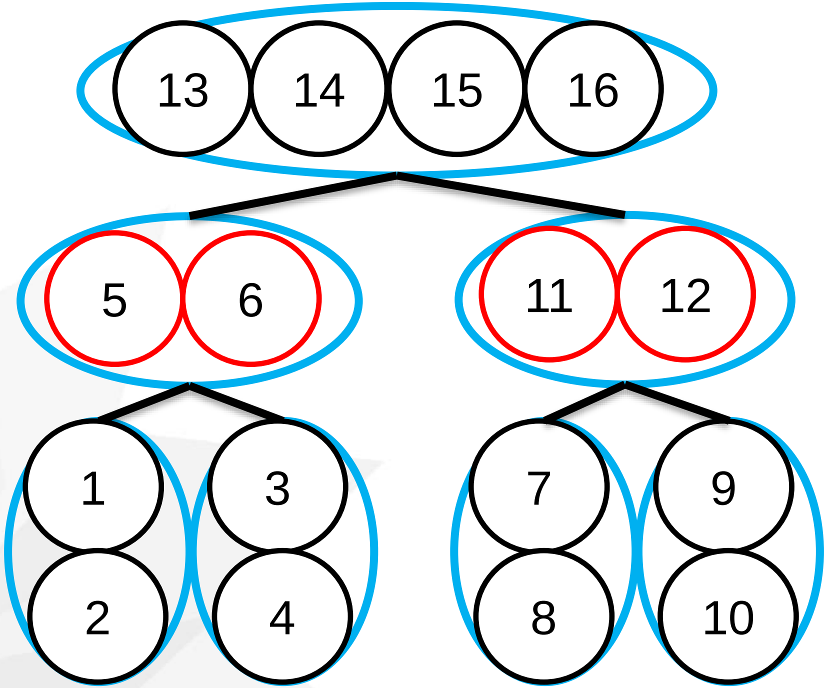


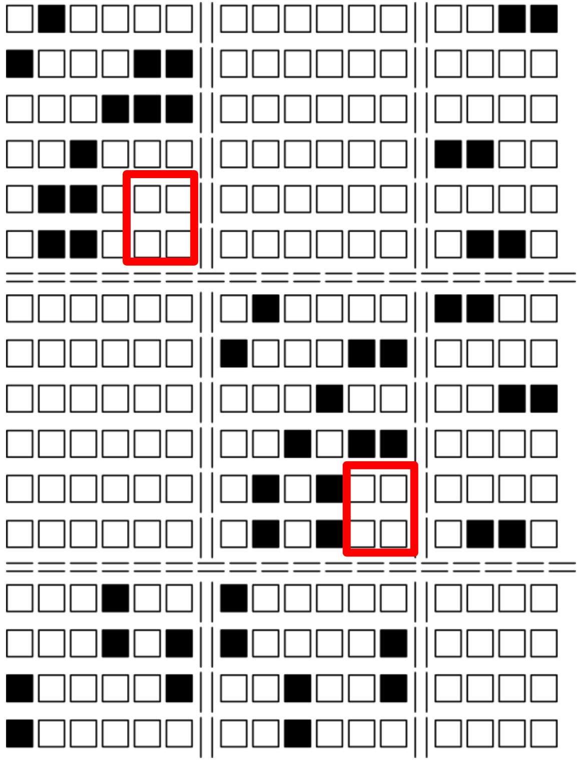


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

For other nodes, do the same process.

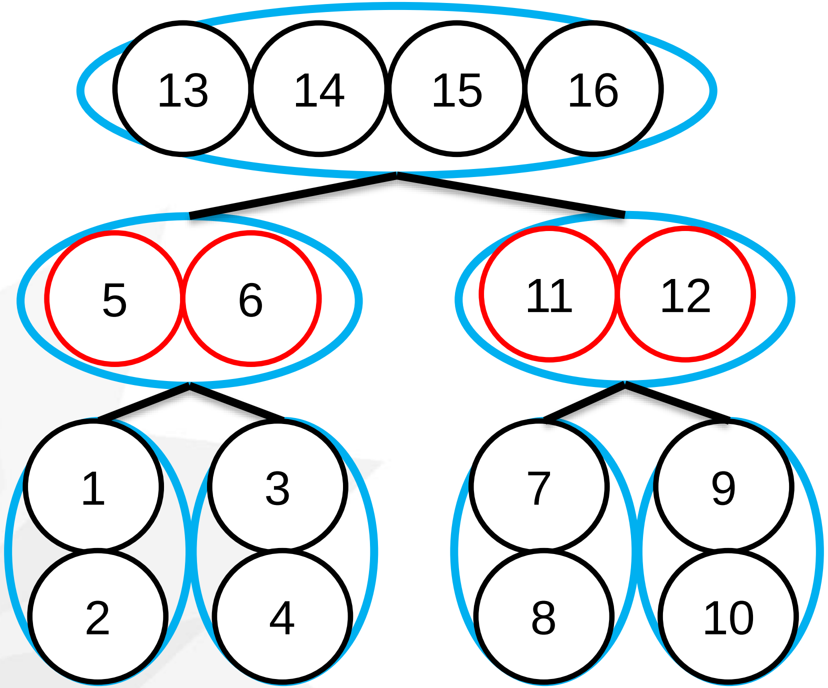


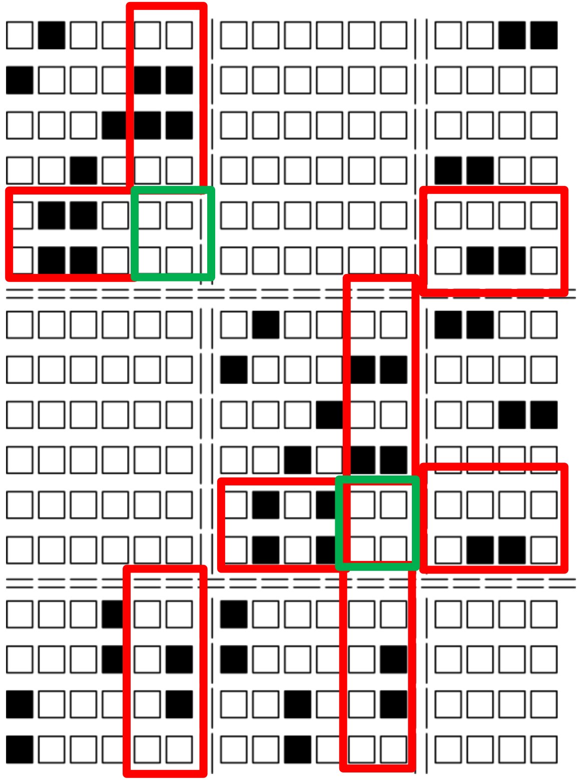


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

For other nodes, do the same process.

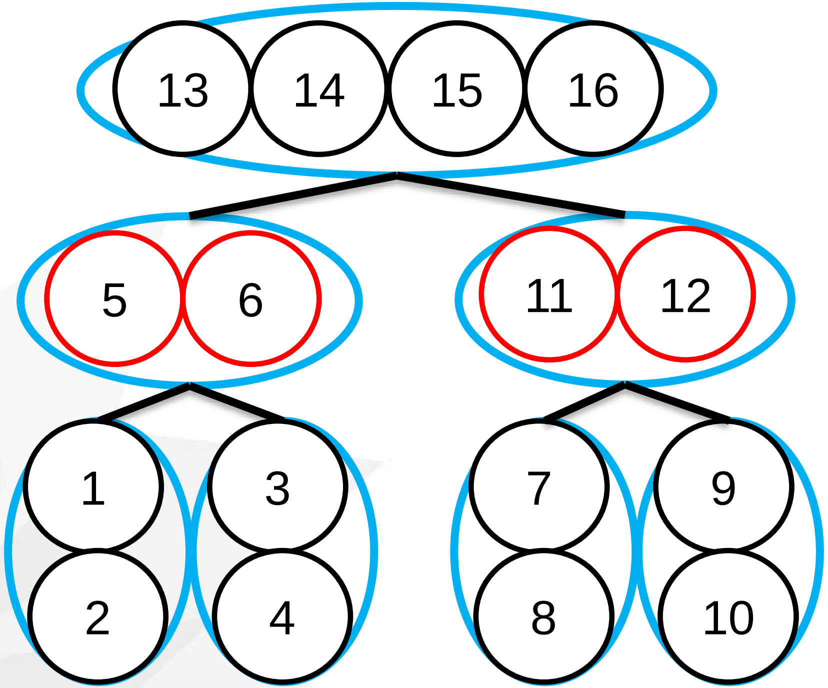


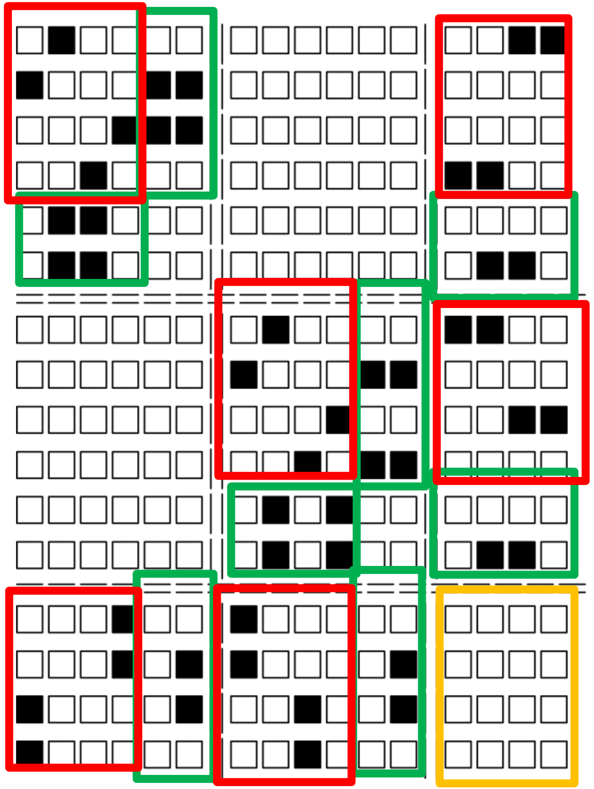


Supernodal Floyd-Warshall

Parallel SuperFW algorithm

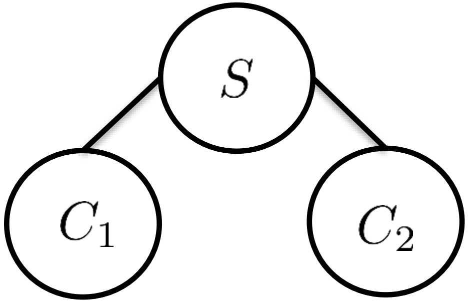
For other nodes, do the same process.





Asymptotic analysis

Asymptotic work



For the work complexity, think about the simplest elimination tree which is above.

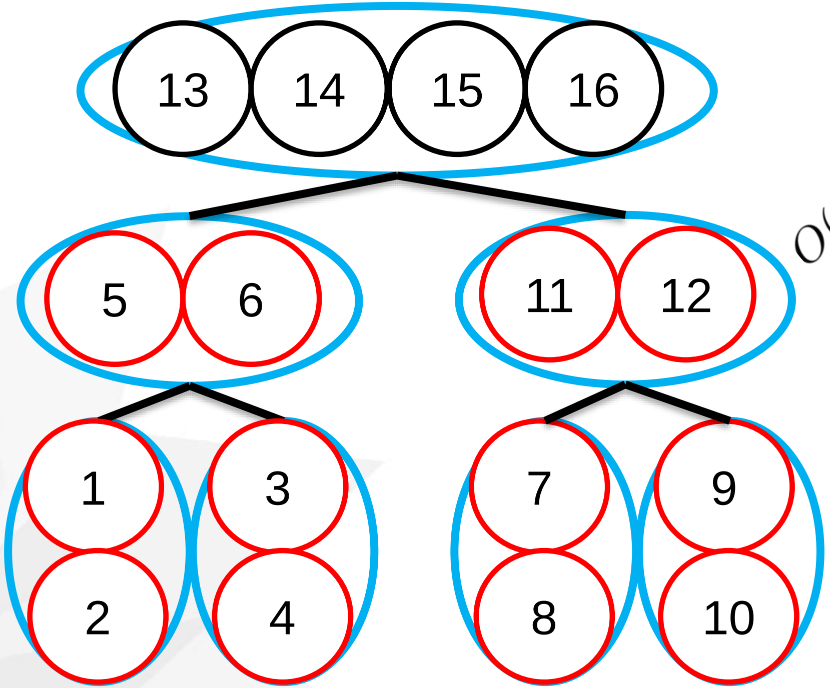
For stage 2 of C_1, C_2, its work complexity is $O((|V|-|S|)^2|S|)$.
Since, it will calculate $O(|V|-|S|)$ intermediaries over $O((|V|-|S|)|S|)$elements.

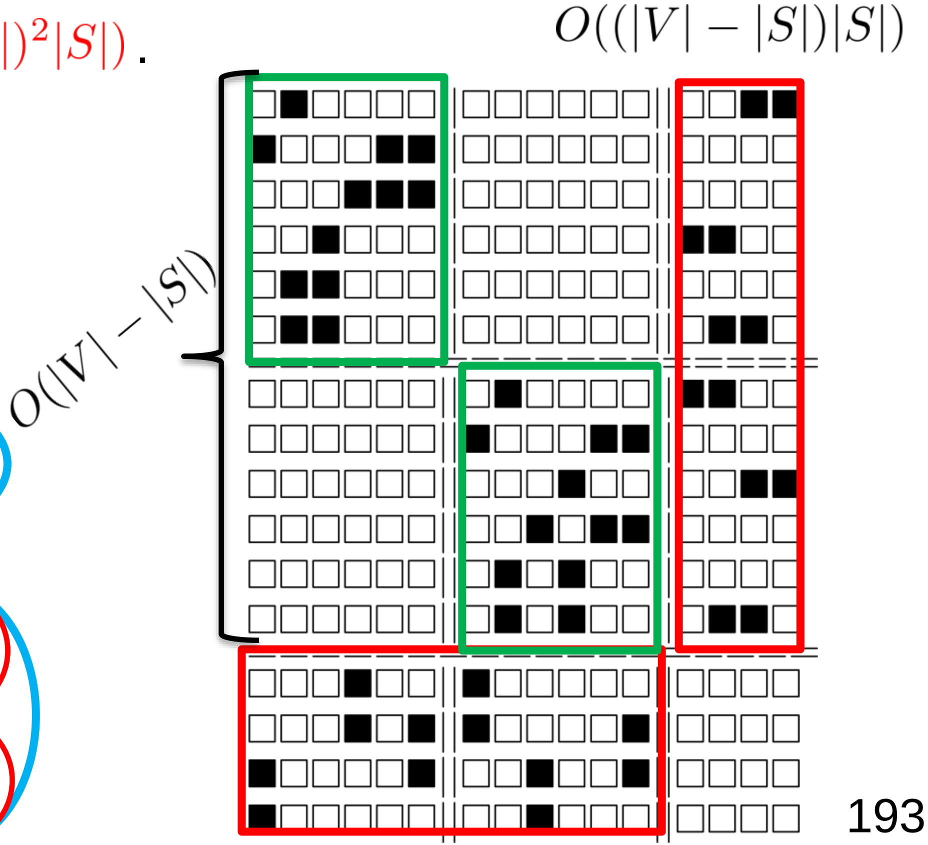
For stage 3 of C_1, C_2, its work complexity is $O((|V|-|S|)|S|^2)$.
Since, it will calculate $O(|V|-|S|)$ intermediaries over $O(|S|^2)$elements.

Asymptotic analysis

Asymptotic work

For stage 2 of C_1, C_2 , takes $O((|V|-|S|)^2|S|)$.

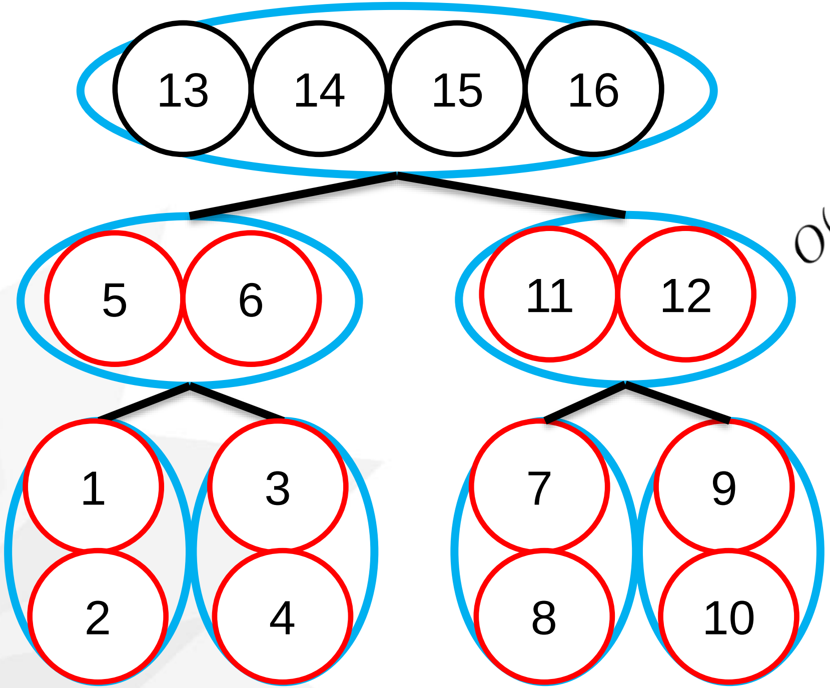


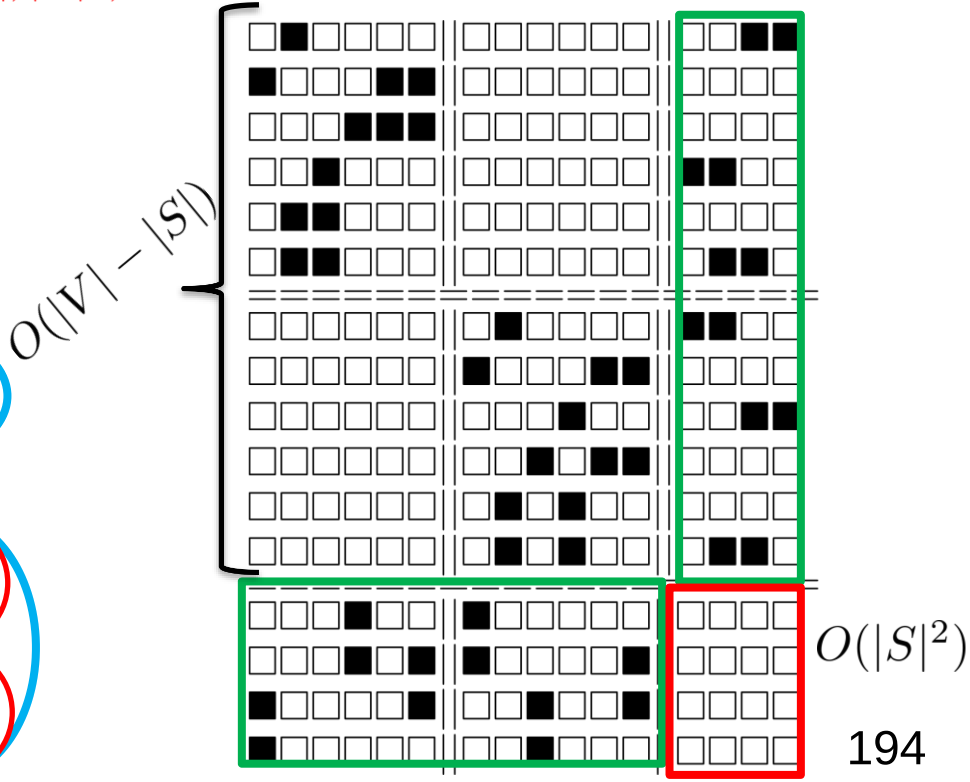


Asymptotic analysis

Asymptotic work

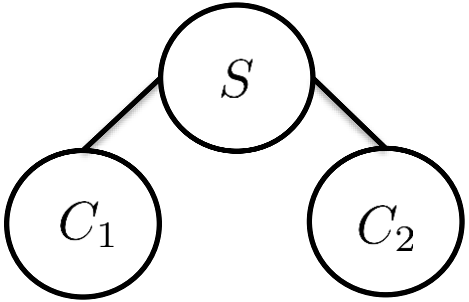
For stage 3 of C_1, C_2 , takes $O((|V| - |S|)|S|^2)$.





Asymptotic analysis

Asymptotic work



For stage 1 of S, its work complexity is $O(|S|^3)$.
Since, it will calculate $O(|S|)$ intermediaries over $O(|S|^2)$ elements.

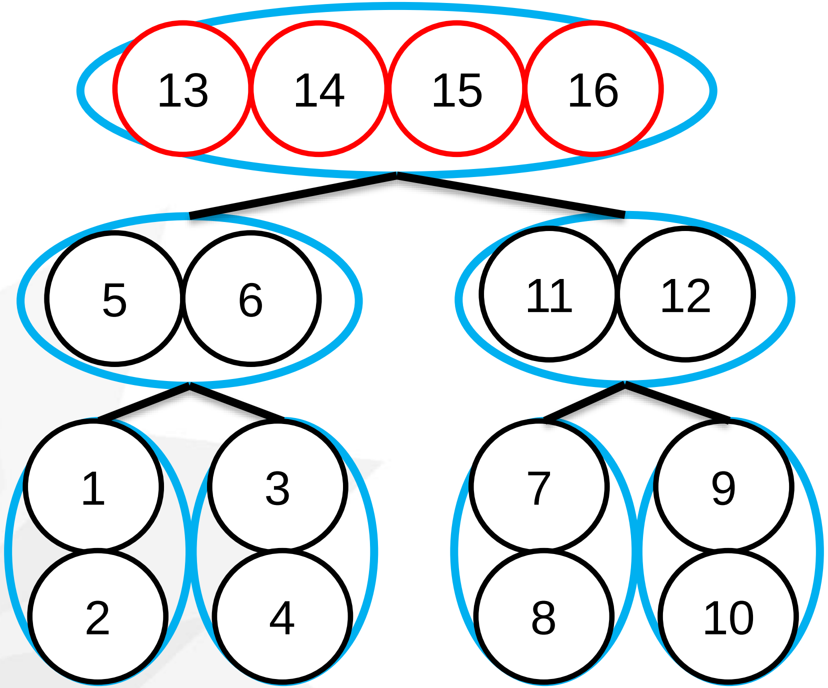
For stage 2 of S, its work complexity is $O((|V|-|S|)|S|^2)$.
Since, it will calculate $O(|S|)$ intermediaries over $O((|V|-|S|)|S|)$ elements.

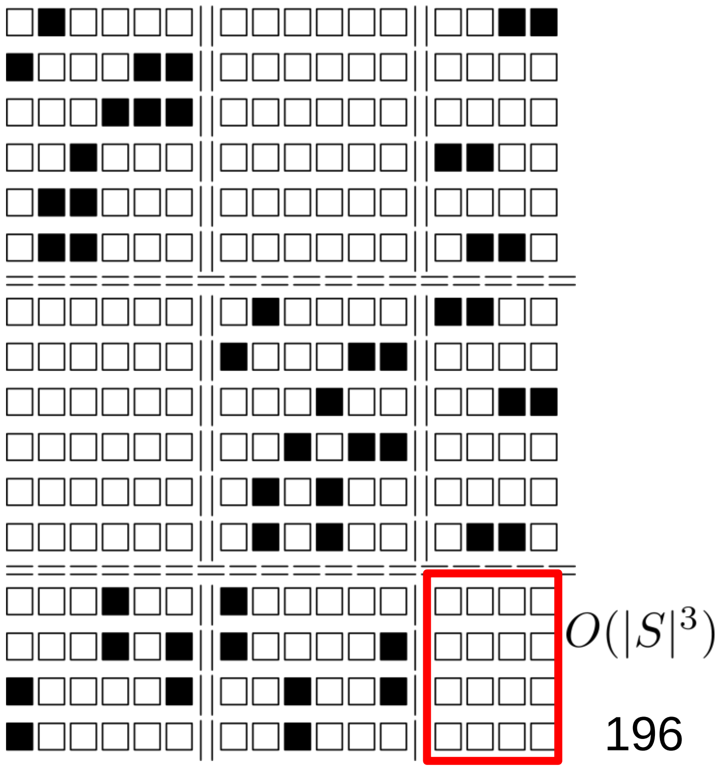
For stage 3 of S, its work complexity is $O((|V|-|S|)^2|S|)$.
Since, it will calculate $O(|S|)$ intermediaries over $O((|V|-|S|)^2)$ elements.

Asymptotic analysis

Asymptotic work

For stage 1 of S, takes $O(|S|^3)$.

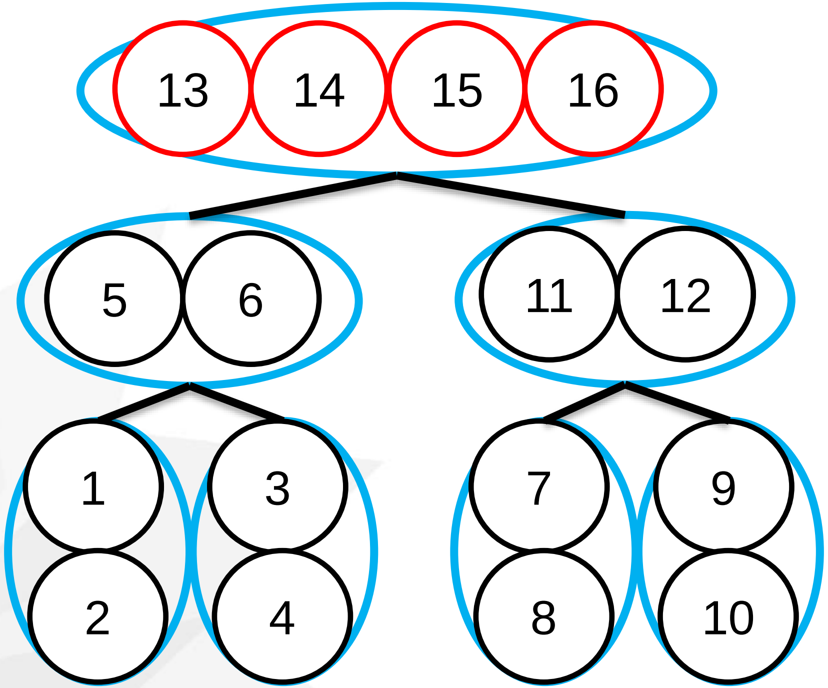


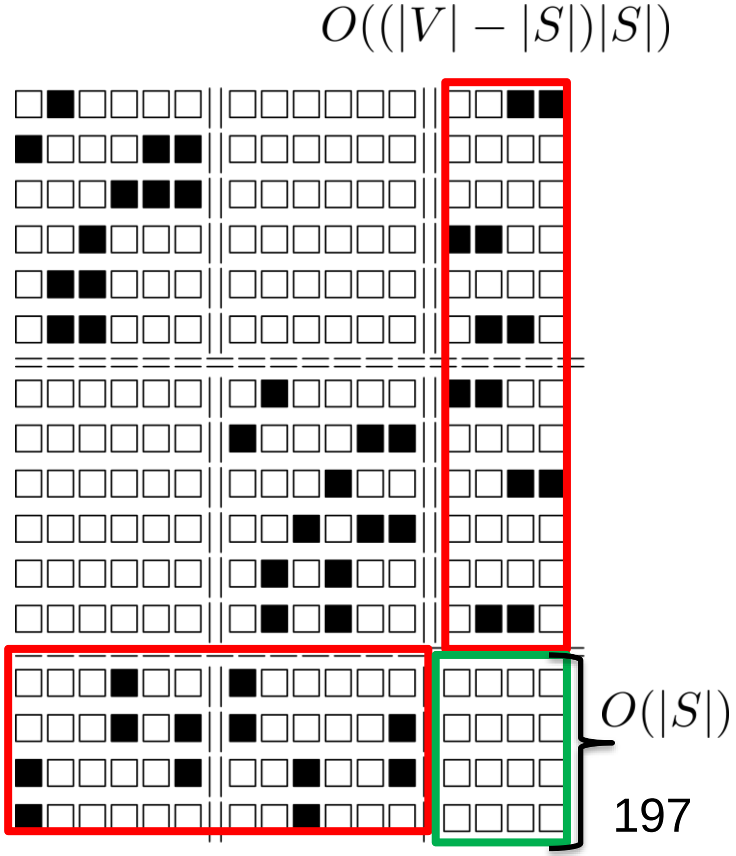


Asymptotic analysis

Asymptotic work

For stage 2 of S, takes $O((|V|-|S|)|S|^2)$.

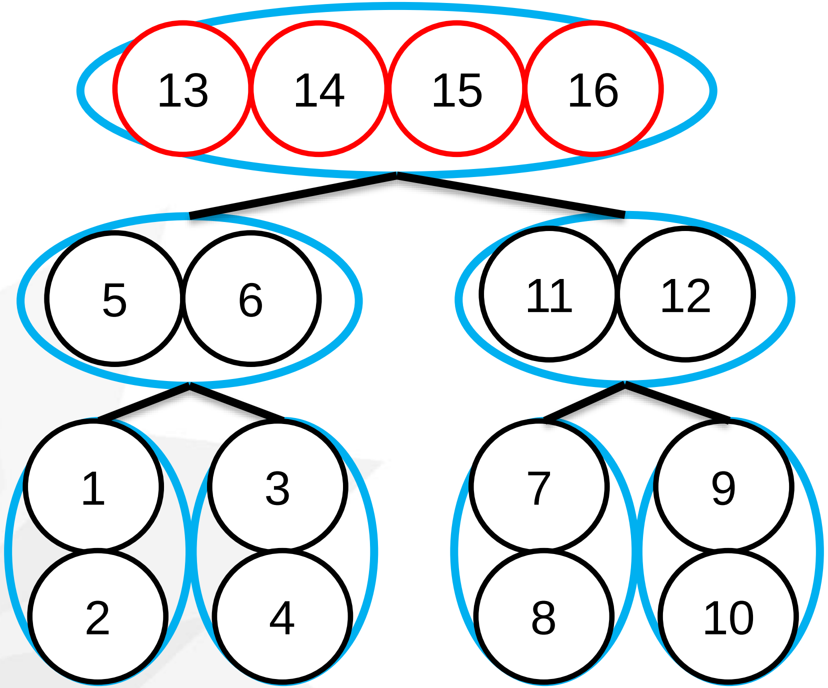


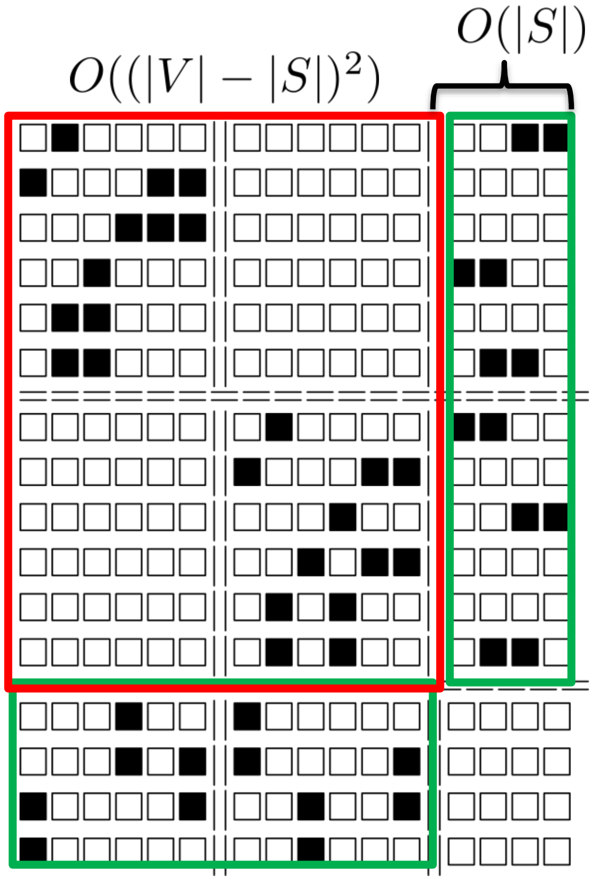


Asymptotic analysis

Asymptotic work

For stage 3 of S, takes $O((|V|-|S|)^2|S|)$.





Asymptotic analysis

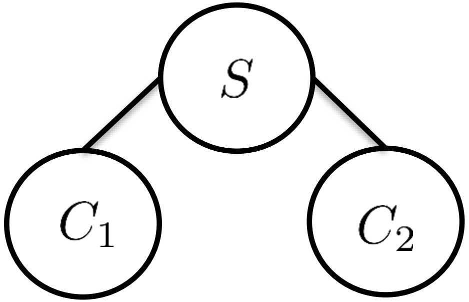
Asymptotic work

Stage	At	Complexity
Stage 2	C_1, C_2	$O((\lvert V\rvert - \lvert S\rvert)^2 \lvert S\rvert)$
Stage 3	C_1, C_2	$O((\lvert V\rvert - \lvert S\rvert) \lvert S\rvert^2)$
Stage 1	S	$O(\lvert S\rvert^3)$
Stage 2	S	$O((\lvert V\rvert - \lvert S\rvert) \lvert S\rvert^2)$
Stage 3	S	$O((\lvert V\rvert - \lvert S\rvert)^2 \lvert S\rvert)$
Total		$O(\lvert V\rvert^2 \lvert S\rvert)$

Notice that $|S| \ll |V|$.

Asymptotic analysis

Asymptotic work

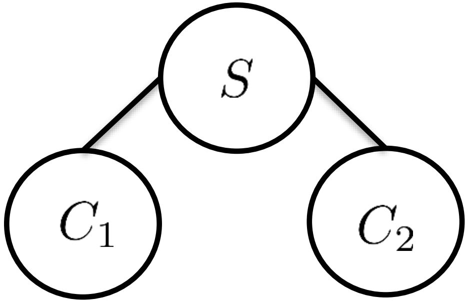


Nested dissection forms the elimination tree by recursion.
Stage 1 of C_1, C_2 can be calculated recursively.
Therefore, total work complexity is formula below.
As a result, work complexity is $O(|V|^2|S|)$.

$$O(|V|^2|S(V)|) + 2 * O(|V|^2/4|S(V/2)|) + 4 * O(|V|^2/4^2|S(V/2^2)|) + \cdots \leq$$
$$O(|V|^2|S(V)|)+2*O(|V|^2/4|S(V)|)+4*O(|V|^2/4^2|S(V)|)+\cdots = 2O(|V|^2|S(V)|)$$

Asymptotic analysis

Asymptotic work with parallel processors



For a naive analysis for parallelism, this paper used a parallel random access memory(PRAM) model. In this model, all processors can access a memory location simultaneously, and only one processor can write at a location at a time(CREW). It’s well known model for parallel algorithm(theorical model).

Asymptotic analysis

Asymptotic work with parallel processors

Algorithm 3 The SuperFw algorithm

```
n_s := Number of supernodes
function SuperFw(G = (V,E)):
    for k = {1, 2..., n_s} do:
Diagonal Update
        A(k,k) ← Floyd-Warshall(A(k,k))
Panel Update
        for i ∈ 𝒜(k) ∪ 𝒟(k) do
            A(i,k) ← A(i,k) ⊕ A(i,k) ⊗ A(k,k)
            A(k,i) ← A(k,i) ⊕ A(k,k) ⊗ A(k,i)
MinPlus Outer Product
        for (i,j) ∈ {𝒜(k) ∪ 𝒟(k)} × {𝒜(k) ∪ 𝒟(k)} do:
            A(i,j) ← A(i,j) ⊕ A(i,k) ⊗ A(k,j)
```

Figure 16. The supernodal Floyd-Warshall algorithm

Parallelizable parts can be calculated In $O(1)$ by $O(|V|^2)$ processors.

$\mathcal{D}(k) \times \mathcal{D}(k)$

$\mathcal{A}(k) \times \mathcal{D}(k)$

$\mathcal{D}(k) \times \mathcal{A}(k)$

$\mathcal{A}(k) \times \mathcal{A}(k)$

Asymptotic analysis

Asymptotic work with parallel processors

Algorithm 3 The SuperFw algorithm

n_s := Number of supernodes
function SuperFw($G=(V,E)$):
  for $k=\{1, 2\ldots,n_s\}$ **do**:
  Diagonal Update
    $A(k,k) \leftarrow$ Floyd-Warshall($A(k,k)$)
  Panel Update
    for $i \in \mathcal{A}(k) \cup \mathcal{D}(k)$ **do**
      $A(i,k) \leftarrow A(i,k) \oplus A(i,k) \otimes A(k,k)$
      $A(k,i) \leftarrow A(k,i) \oplus A(k,k) \otimes A(k,i)$
  MinPlus Outer Product
    for $(i,j) \in \boxed{\{\mathcal{A}(k) \cup \mathcal{D}(k)\} \times \{\mathcal{A}(k) \cup \mathcal{D}(k)\}}$ **do**:
      $A(i,j) \leftarrow A(i,j) \oplus A(i,k) \otimes A(k,j)$

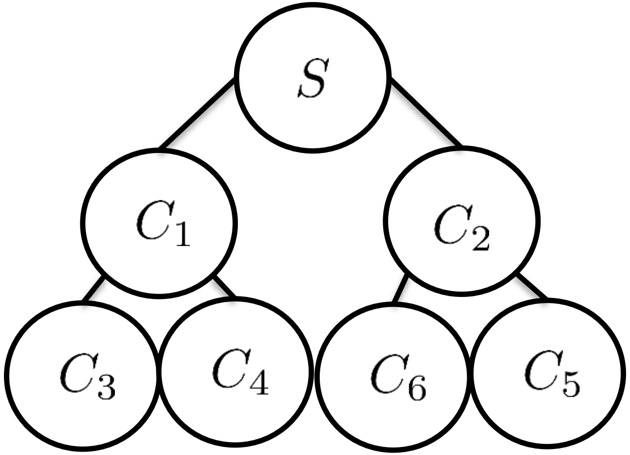
$$\mathcal{D}(k) \times \mathcal{D}(k)$$
$$\mathcal{A}(k) \times \mathcal{D}(k)$$
$$\mathcal{D}(k) \times \mathcal{A}(k)$$
$$\mathcal{A}(k) \times \mathcal{A}(k)$$

This is only part that can't be parallelized which will calculate by tree reduction order.

Figure 16. The supernodal Floyd-Warshall algorithm

Asymptotic analysis

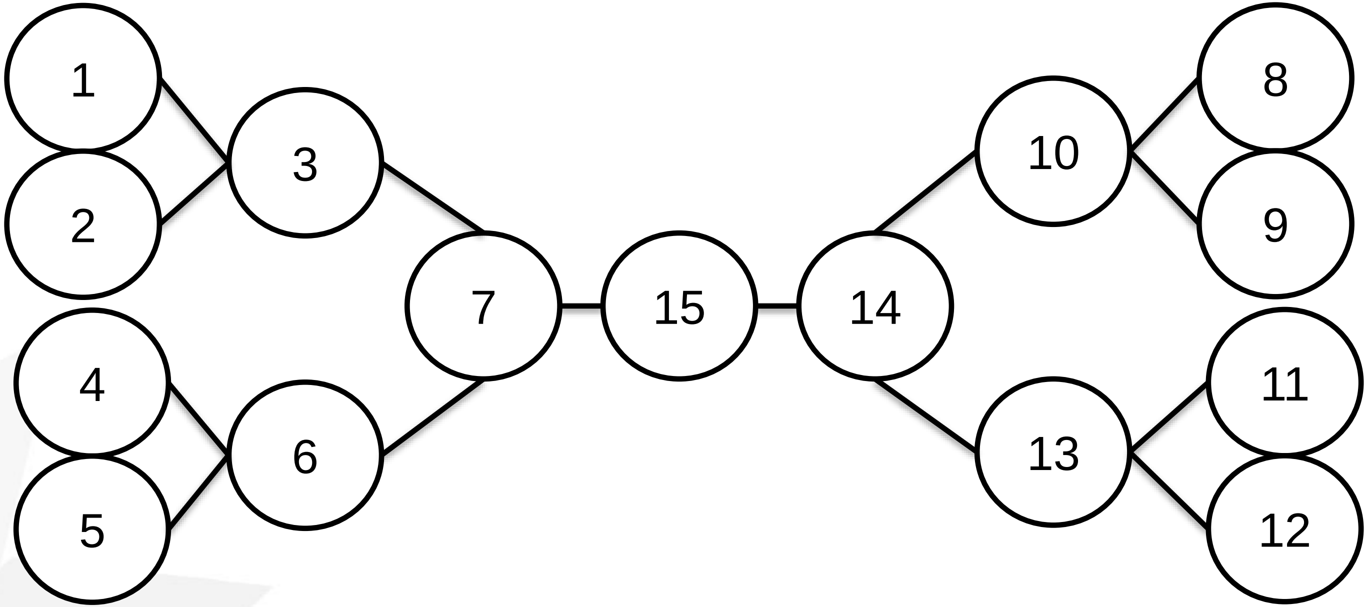
Asymptotic work with parallel processors



If it assume that tree is balanced. Which means $|C_1| \approx |C_2|$.
Then for compute stage 3 of C_1, C_2 needs $O(S(\frac{|V|}{2}))$ for the most outer loop.
For compute stage 3 of C_3, C_4, C_5, C_6 needs $O(2 * S(\frac{|V|}{4}))$ for the most outer loop.
It's easy to think the work complexity as how many intermediaries needs.
Detail explain is follow.

Asymptotic analysis

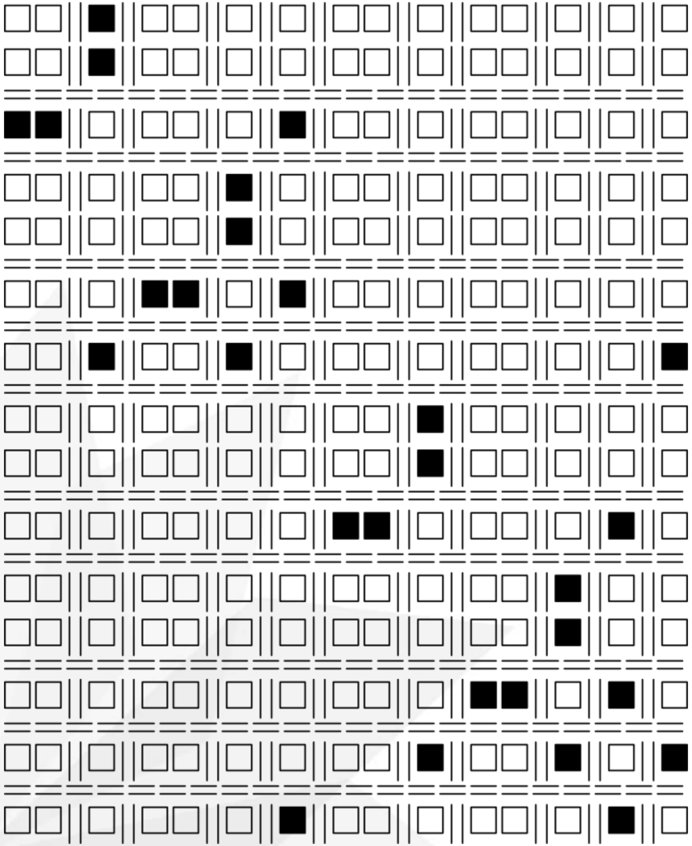
Asymptotic work with parallel processors

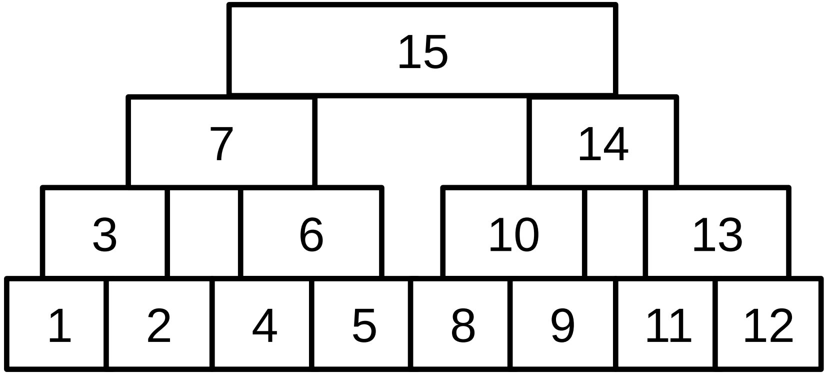


For the simple explanation, example graph with order above will be used.

Asymptotic analysis

Asymptotic work with parallel processors

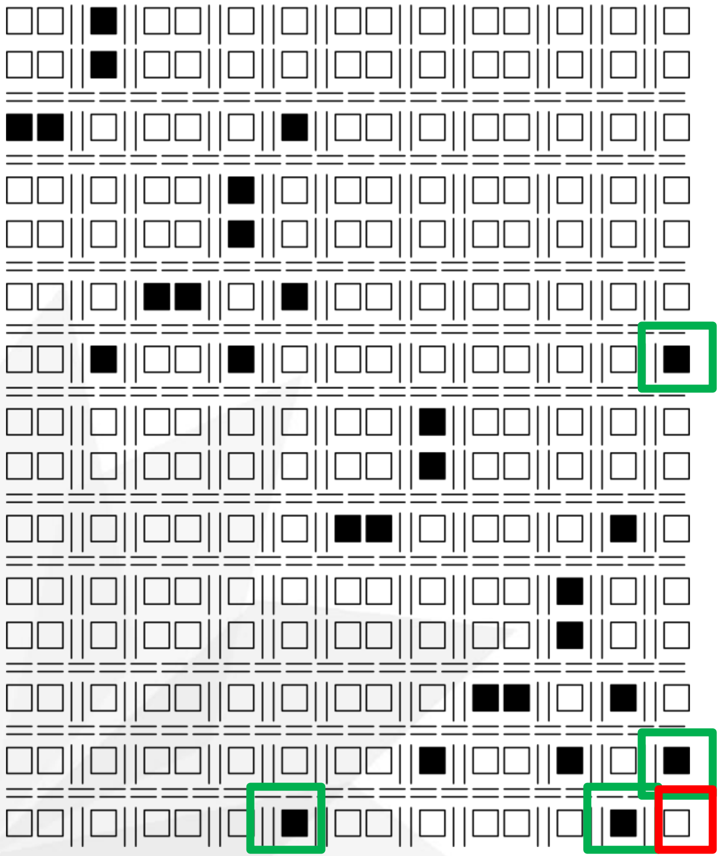


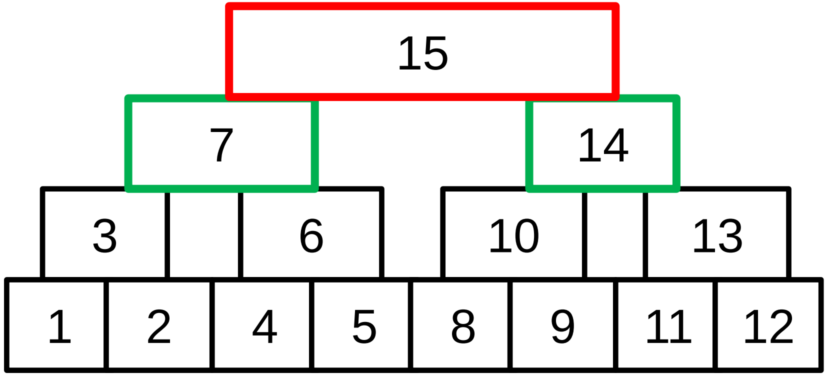


In this analysis it will consider only computation over ancestors since it’s the only parts need a time.

Asymptotic analysis

Asymptotic work with parallel processors

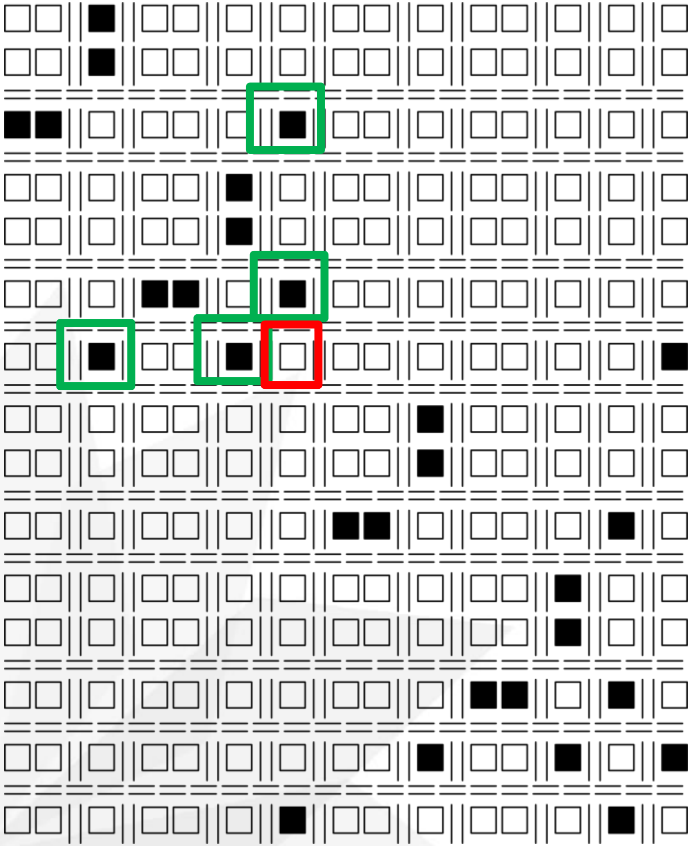


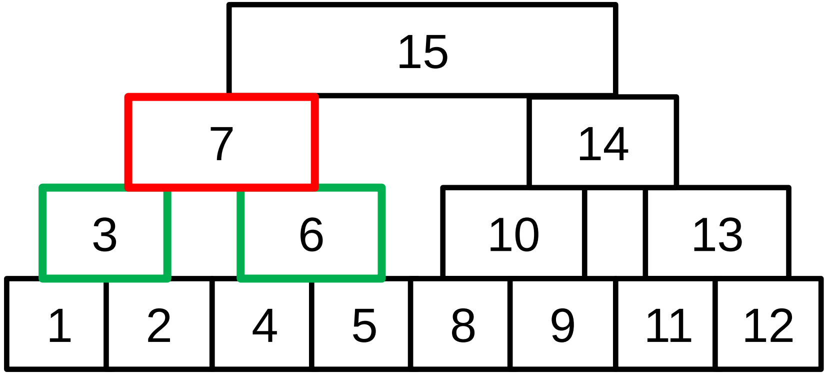


For (7,14) to (15),
It needs only one tree reduction operation.
It takes $O(S(V/2))$.
Notice that 7,14 is also separator.

Asymptotic analysis

Asymptotic work with parallel processors

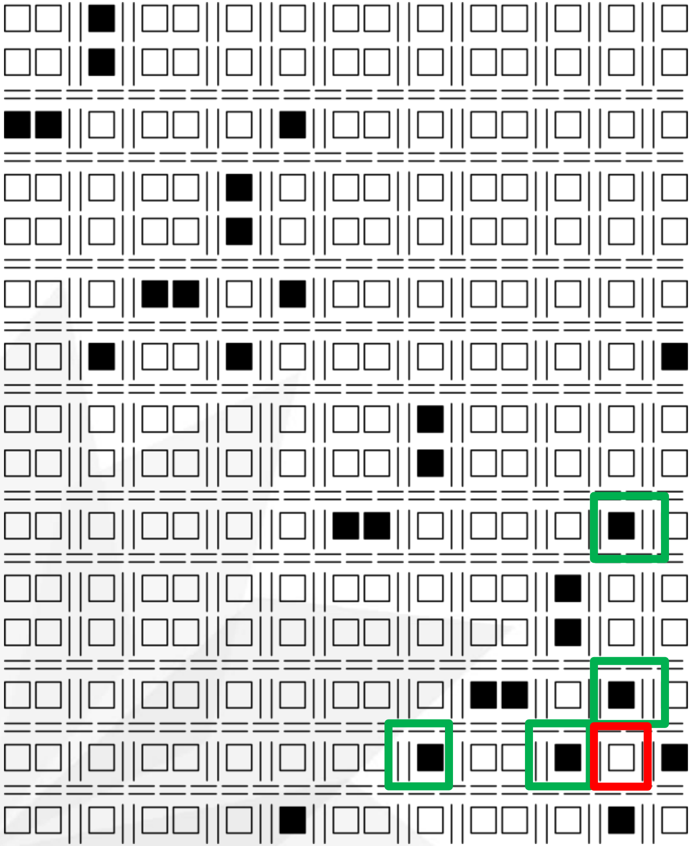


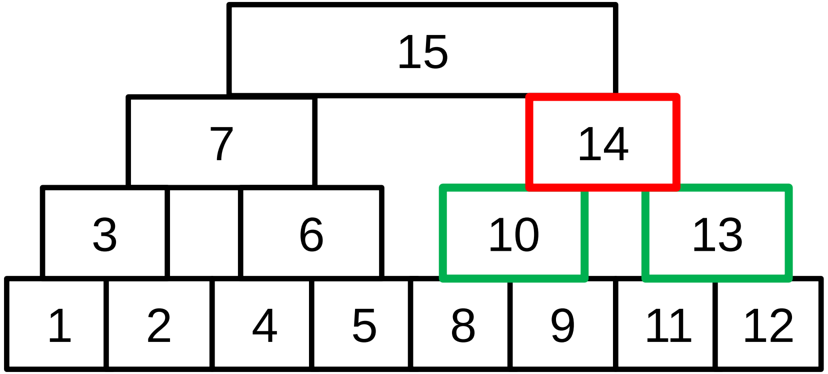


For (3,6) to (7),
It needs only one tree reduction operation.
It takes $O(S(V/4))$.
Notice that 3,6 is also separator.

Asymptotic analysis

Asymptotic work with parallel processors

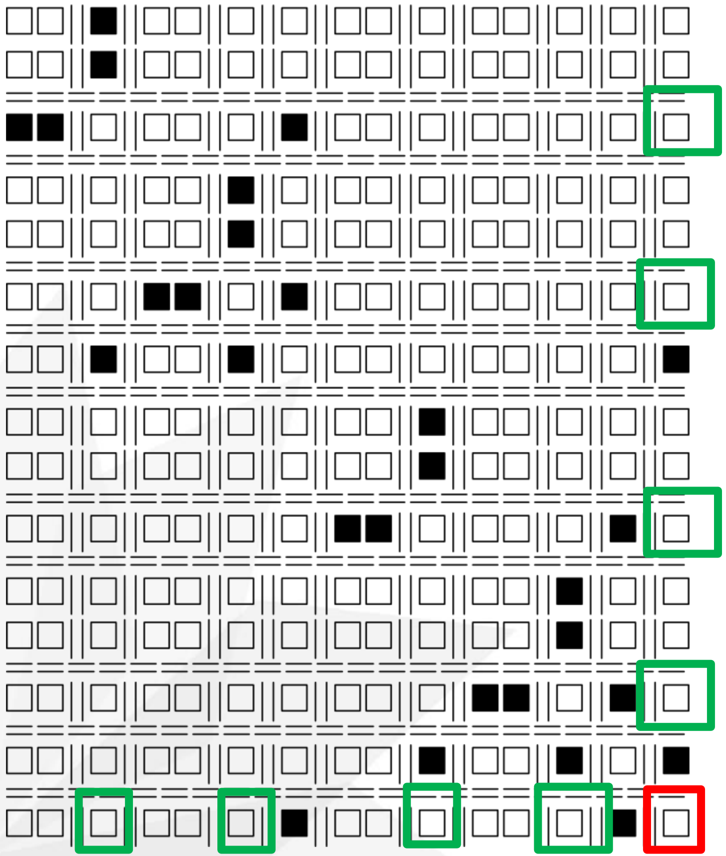


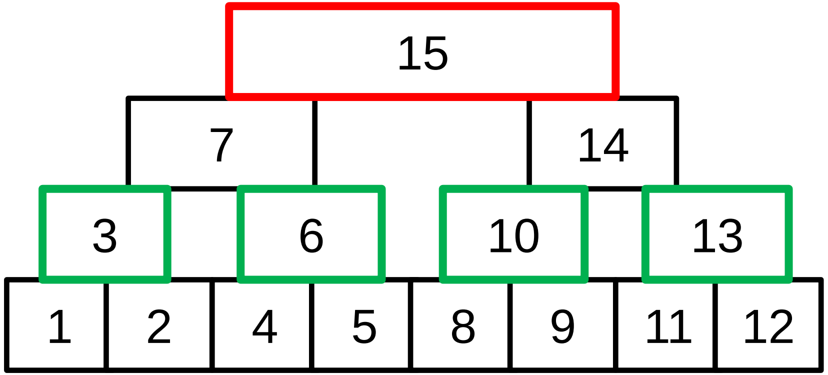


For (10,13) to (14),
It needs only one tree reduction operation.
It takes $O(S(V/4))$.
Notice that 10,13 is also separator.

Asymptotic analysis

Asymptotic work with parallel processors

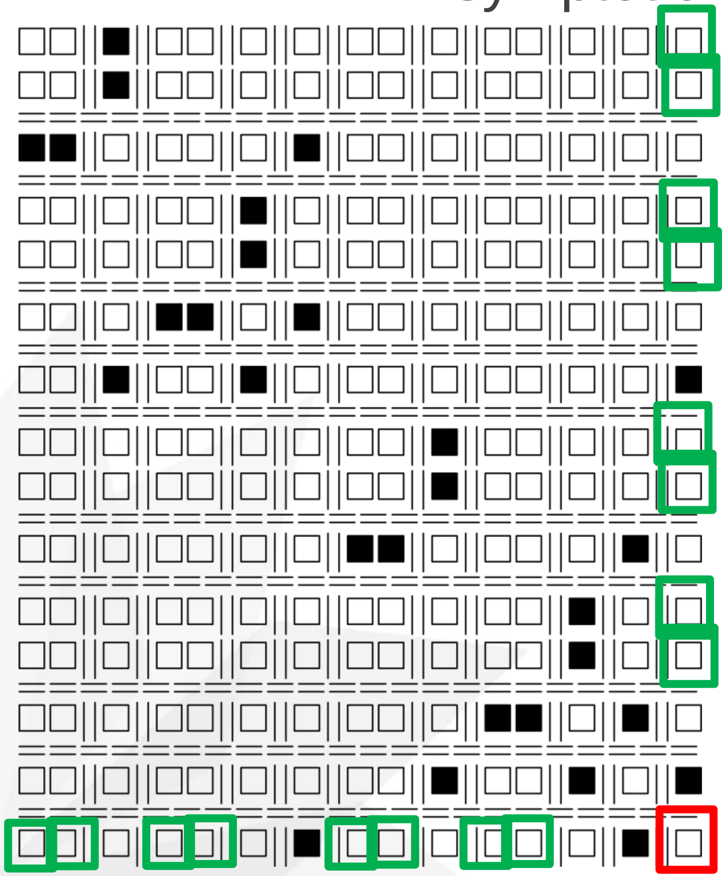


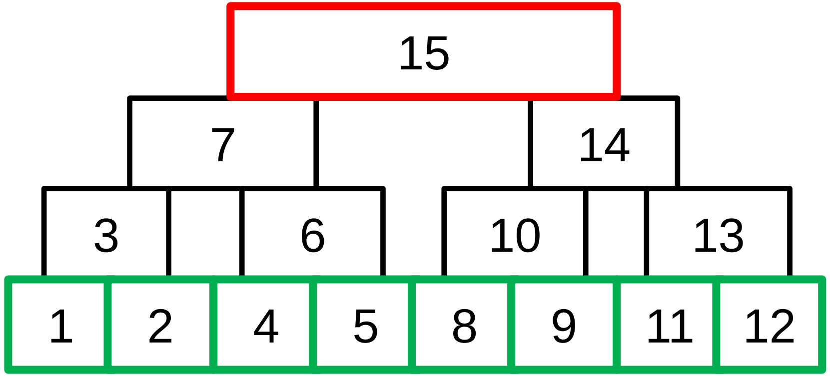


For (3,6), (10,13) to (15),
It needs two tree reduction operation.
It takes $2 \times O(S(V/4))$.
Notice that 3,6,10,13 is also separator.

Asymptotic analysis

Asymptotic work with parallel processors

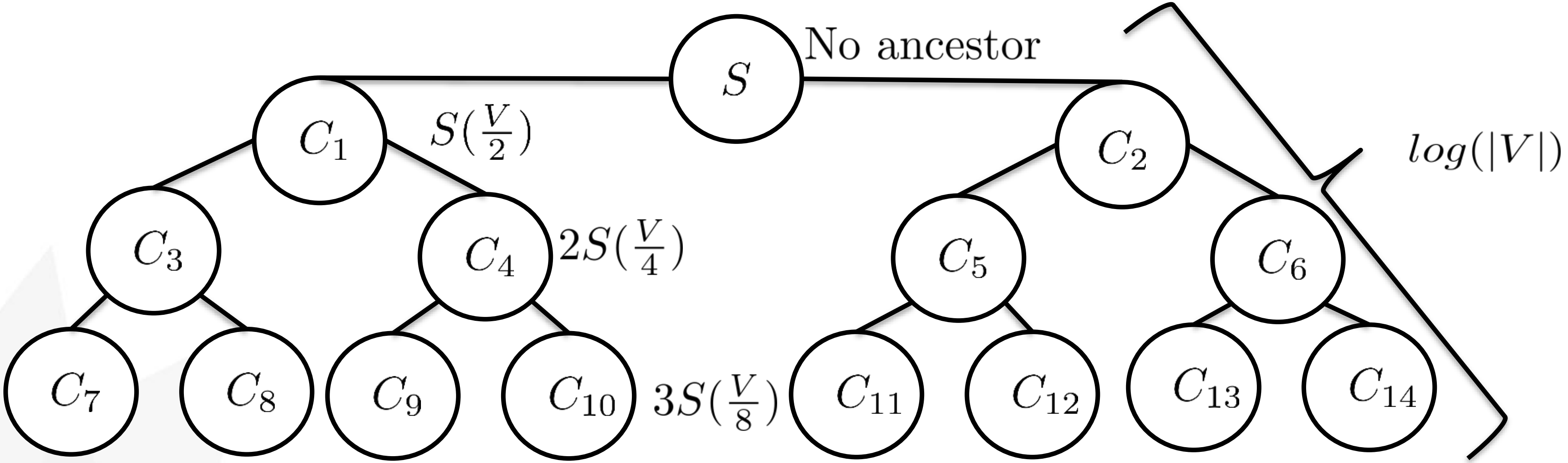




If there is more depth at e-tree.
For (1,2), (4,5),(8,9),(11,12) to (15),
It needs three tree reduction operation.
It takes $3 \times O(S(V/8))$.
You can see the pattern.

Asymptotic analysis

Asymptotic work with parallel processors



As a result, work complexity with PRAM model is as bellow.

$$D(|V|) = \Sigma_{k=1}^{log|V|}(k \times S(|V|/2^k)) \leq \Sigma_{k=1}^{log|V|}(log|V| \times S(|V|)) = S(|V|) \times (log|V|)^2$$

Asymptotic analysis

Comparison

Algorithm	Work complexity	PRAM complexity	Max-speedup
Dijkstra	$O(\|V\|^2 log\|V\| + \|V\|\|E\|)$	$O(\|V\| log\|V\| + \|E\|)$	$O(\|V\|)$
BlockedFW	$O(\|V\|^3)$	$O(\|V\|)$	$O(\|V\|^2)$
SuperFW	$O(\|V\|^2\|S\|)$	$O(\|S\|(log\|V\|)^2)$	$O(\frac{\|V\|^2}{(log\|V\|)^2})$
PathDoubling	$O(\|V\|^3)$	$O(log\|V\|)$	$O(\frac{\|V\|^3}{log\|V\|})$

PathDoubling is another extension of Floyd-Warshall algorithm.
Which is the fastest algorithm ever known for parallel case.
From the theorical background, it is the shortest way to find cost of path.

Results

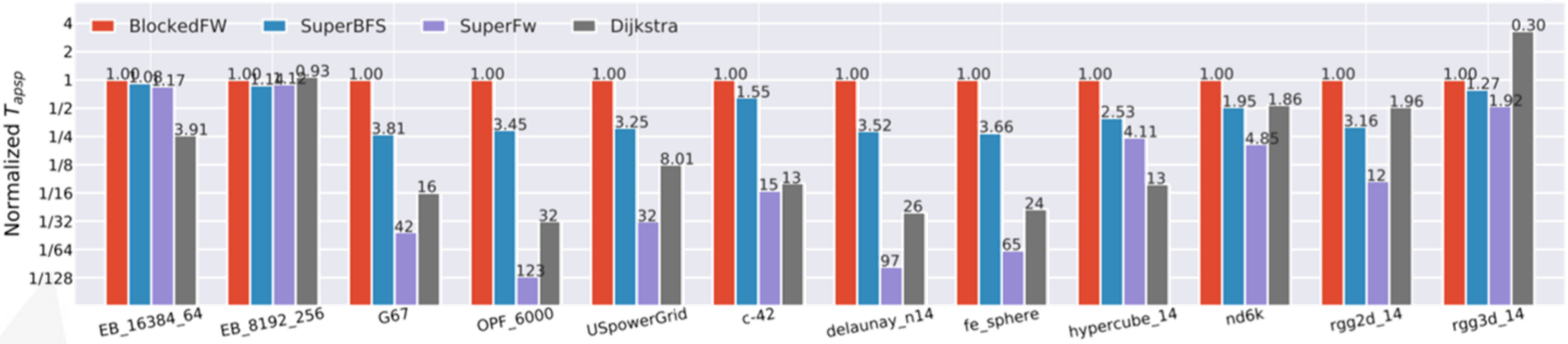
Experiments has been on a shared memory system that contains 32 cores as a dual-socket 16 core Intel E5-2698 v3 “Haswell” processors. Each socket has 40-MB shared L3 cache. It has a total of 128 GB DDR4 2133 MHz memory arranged in four 16GB DIMMs per socket. Each core can support two hyper threads, thus 64 threads in total.

Results

Name	Source	n	$\frac{nnz}{n}$	$\frac{n}{\lvert S\rvert}$
USpowerGrid	Power network	4.9e3	2.66	6.2e2
OPF_6000	Power network	2.9e4	9.1	1.4e3
nd6k	3D	1.8e4	383	5.8
oilpan	structural	7.3e4	29.1	1.7e2
finan512	Optimization	7.5e4	7.9	1.5e3
net4-1	Optimization	8.8e4	28	2.9e3
c-42	Optimization	1.0e4	10.58	1.5e2
c-69	Optimization	6.7e4	9.24	2.0e2
lpl1	Optimization	3.2e4	10	4.8e2
onera_dual	Structural	8.5e4	4.9	1.5e2
email-Enron	SNAP	3.7e4	9.9	52
delaunay_n14	DIMACS10	1.6e4	5.99	1.7e2
delaunay_n16	DIMACS10	6.5e4	5.99	1.7e2
fe_sphere	DIMACS10	1.6e4	5.99	8.5e1
luxembourg_osm	DIMACS10	1.1e5	2.1	6.7e3
fe_tooth	DIMACS10	7.8e4	11.6	88
wing	DIMACS10	6.2e4	3.9	1.0e2
t60k	DIMACS10	6.0e4	3.0	1.1e3
G67	Random	1e4	4	5.0e1
EB_8192_256	Barabasi - Albert	8.1e3	256	2.5e0
EB_16384_64	Barabasi - Albert	1.63e4	64	2.6e0
rgg2d_14	Random Geometric	1.63e4	128.17	1.6e1
rgg3d_14	Random Geometric	1.63e4	910	2.57
hypercube_14	hypercube Graph	1.6e4	28	5.0e0

Graphs in use are as above.
There are some graphs from real, some from theorical model.

Results



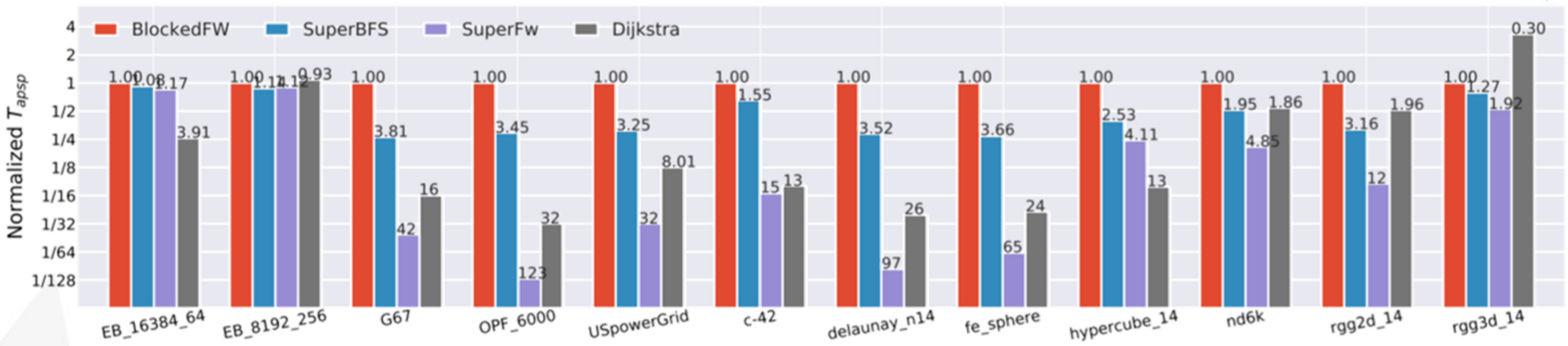
(a) Small Graphs

BlcokedFw means Blocked Floyd-Warshall algorithm using OpenMP.

SuperBFS means Super Floyd-Warshall algorithm that doesn't use Nested dissection ordering. It just uses BFS and order vertices in an order of depth.

Dijkstra means using the Dijkstra's algorithm to solve APSP.

Results

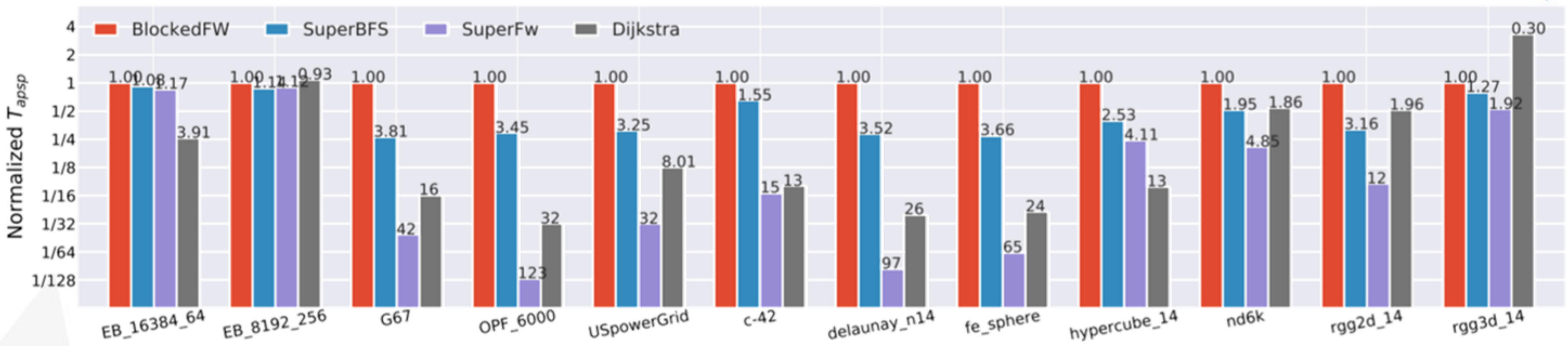


(a) Small Graphs

Supernodal structure’s effect is shown at difference between BlockedFW and SuperBFS.
Supernodal structure effects up to x3.81 at G67

Nested dissection effect is shown at difference between SuperBFS and SuperFW.
Nested dissection effects up to X35.65 at OPF_6000

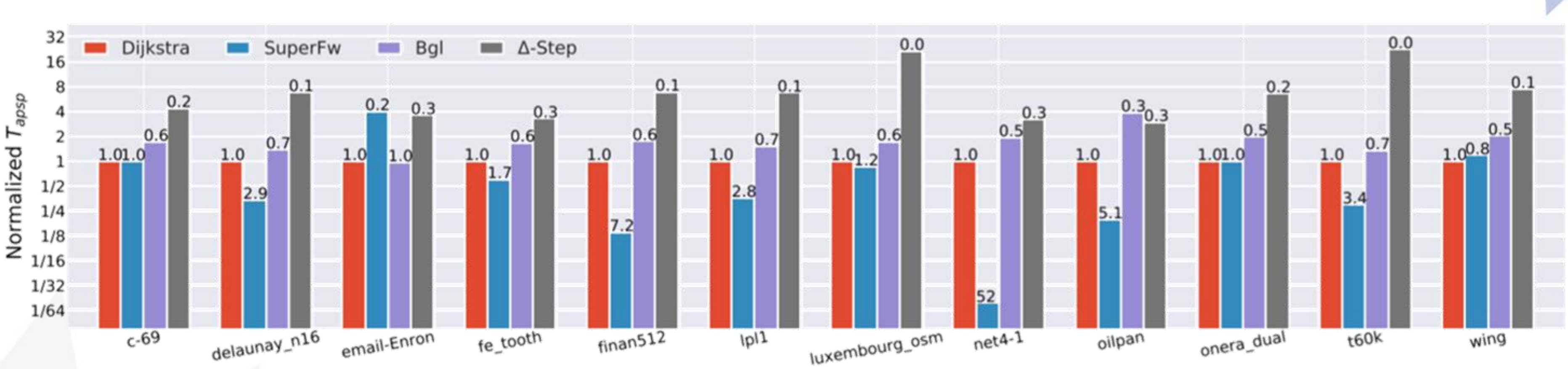
Results



(a) Small Graphs

As a result, SuperFW works pretty nice on small graphs.
There are two excepts which is EB_16384_64 and hypercube_14.
Both graphs doesn’t have a nice cut to make a nested dissection.
Which means that can’t get much improvement for this algorithm.
Weird thing is EB_8192_256 which produced by same algorithm with EB_16384_64, but it shows a nice performance.

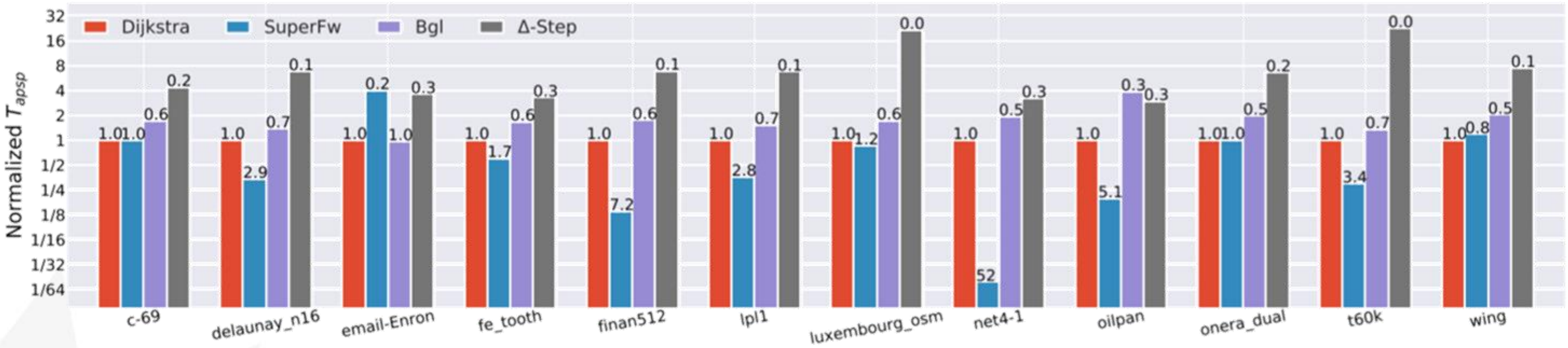
Results



(b) Large Graphs

For large graphs, Bgl and Delta-step algorithm has been experimented instead of SuperBFS and BlockedFW because both shows worse power than SuperFW for every test cases at small graphs.

Results



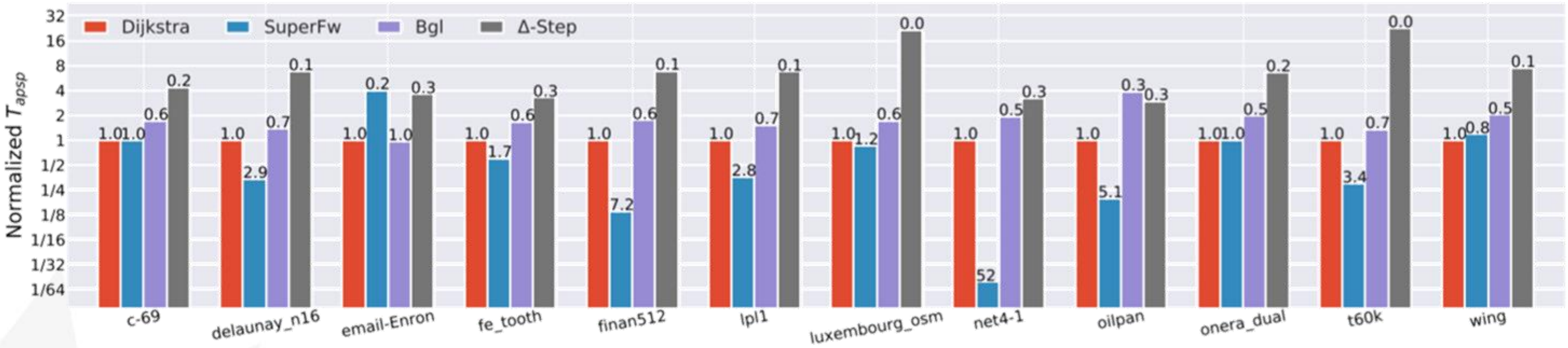
(b) Large Graphs

Bgl is a Dijkstra algorithm implementation by popular Boost Graph Library(BGL).

Delta-step is a delta-step variant of Dijkstra's algorithm.
In this case, its implementation is from Galois Graph library.

This paper doesn't show performance in comparison with the Path Doubling.

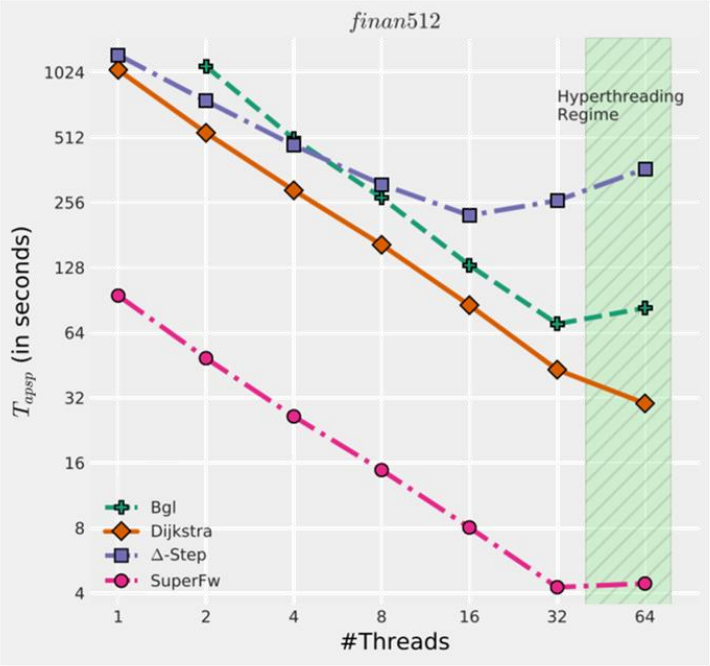
Results



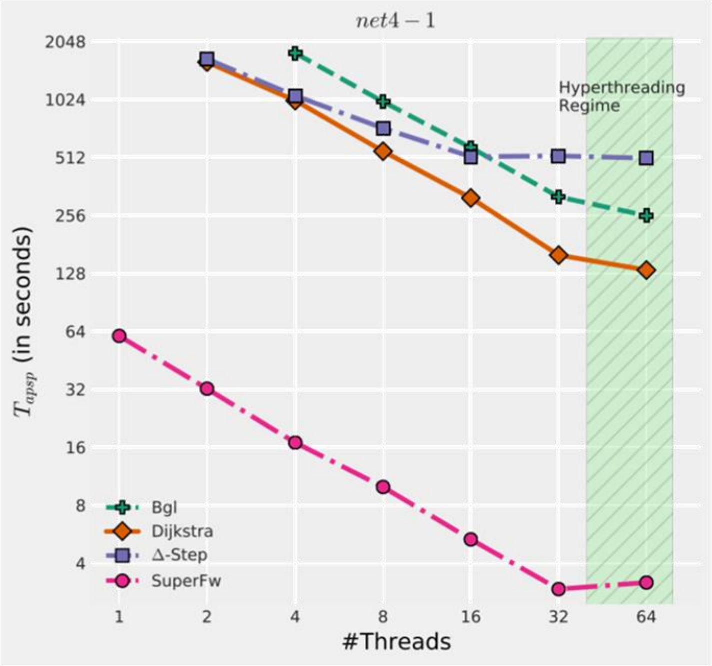
(b) Large Graphs

In most of cases, SuperFW domains performance.
Speedup derivates between x0.2 and x52.
The only bad case is an email-Enron which is email-network.
There is no much information at paper for this graph.☹
However, we can imagine that email network is hard to be bipartite.☺

Results



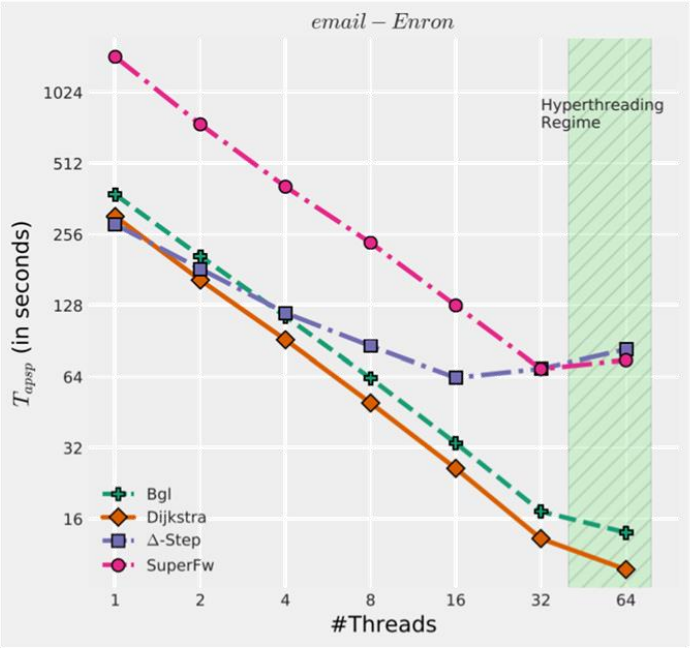
(a) finan512



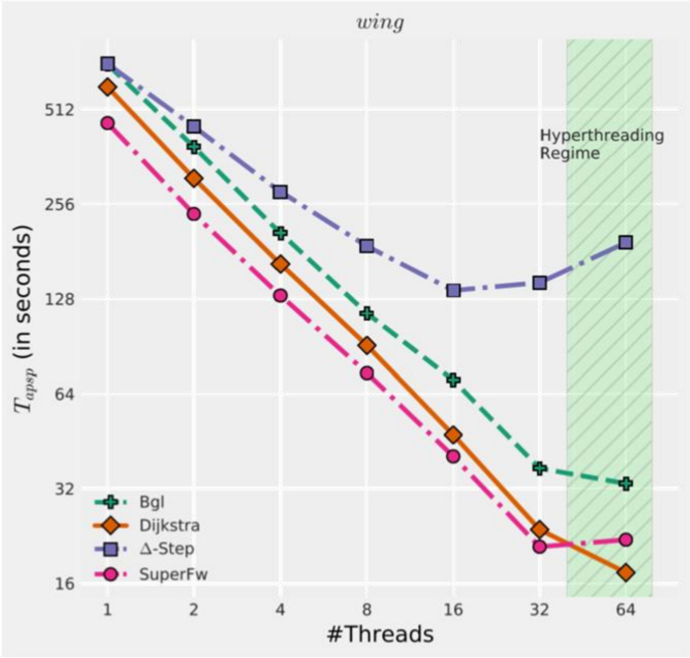
(b) net4-1

For scalability, it scales well until the number of physical threads.
Even in the bad graph(email-Enron), it works fine.

Results



(c) email-Enron



(d) wing

For scalability, it scales well until the number of physical threads.
Even in the bad graph(email-Enron), it works fine.

Thank you

Any comments and questions