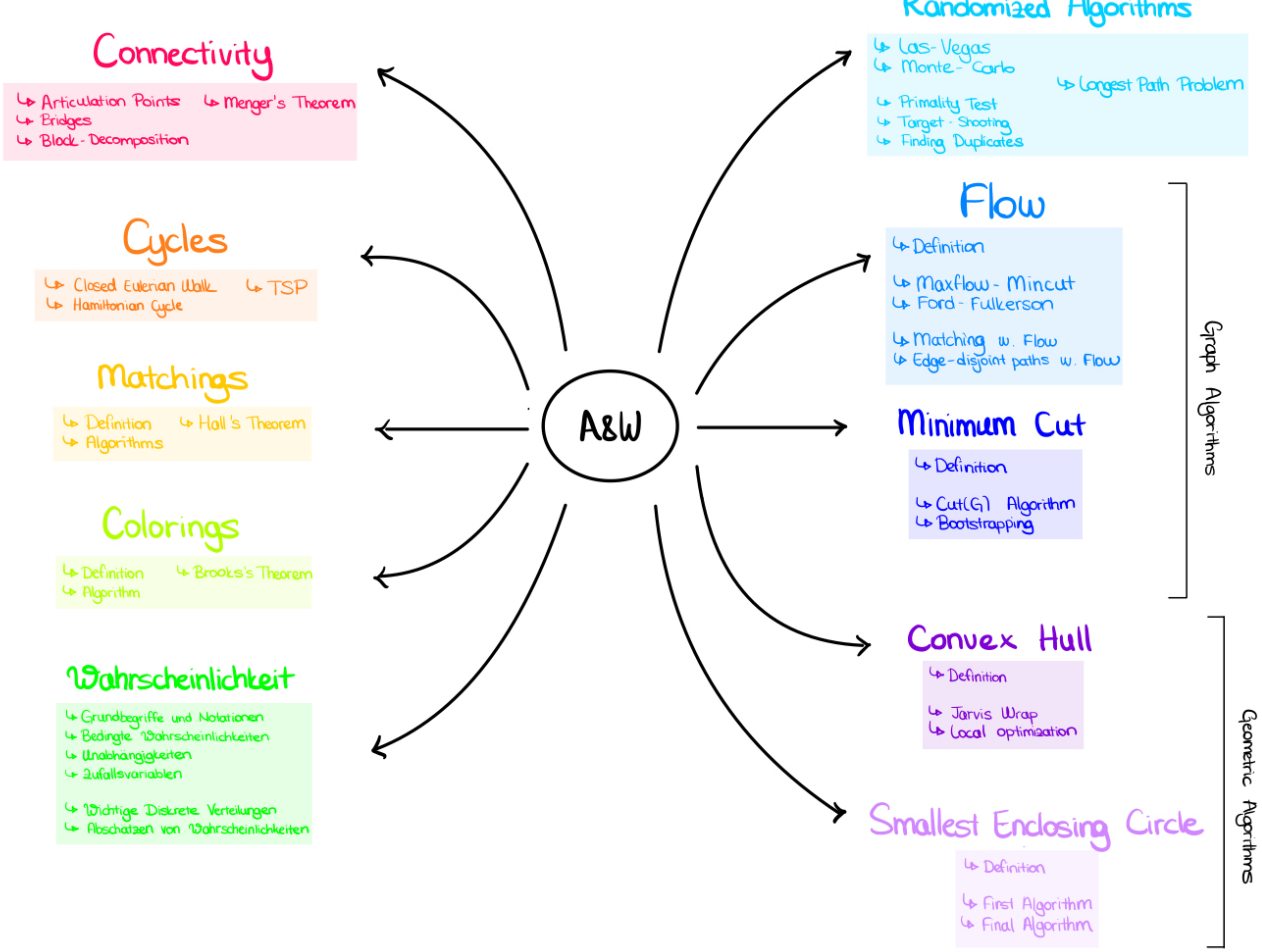
A&W

Exercise Session 4

Matching, TSP II

Nil Ozer

A&W Overview


A&W
Connectivity
Articulation Points
Bridges
Block - Decomposition
Menger's Theorem
Cycles
Closed Eulerian Walk
Hamiltonian Cycle
TSP
Matchings
Definition
Algorithms
Hall's Theorem
Colorings
Definition
Algorithm
Brooks's Theorem
Wahrscheinlichkeit
Grundbegriffe und Notationen
Bedingte Wahrscheinlichkeiten
Unabhängigkeiten
Zufallsvariablen
Wichtige Diskrete Verteilungen
Abschätzen von Wahrscheinlichkeiten
Randomized Algorithms
Las-Vegas
Monte-Carlo
Primality Test
Target - Shooting
Finding Duplicates
Longest Path Problem
Flow
Definition
Maxflow - Mincut
Ford - Fulkerson
Matching w. Flow
Edge-disjoint paths w. Flow
Minimum Cut
Definition
Cut(G) Algorithm
Bootstrapping
Graph Algorithms
Convex Hull
Definition
Jarvis Wrap
Local optimization
Smallest Enclosing Circle
Definition
First Algorithm
Final Algorithm
Geometric Algorithms

Outline

- Minitest II
- Minitest II Discussion

- Matching
- TSP II

Minitest II

Matching

Matching
Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

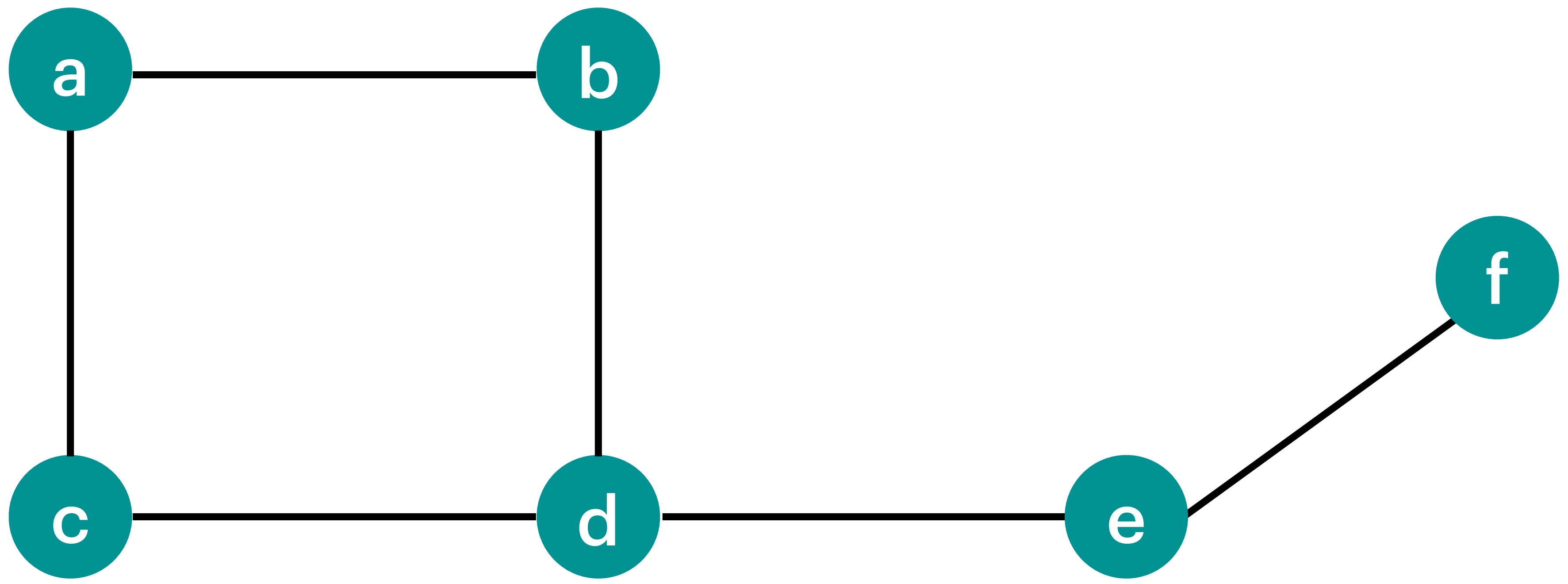
no two edges share common vertices

Matching
Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?

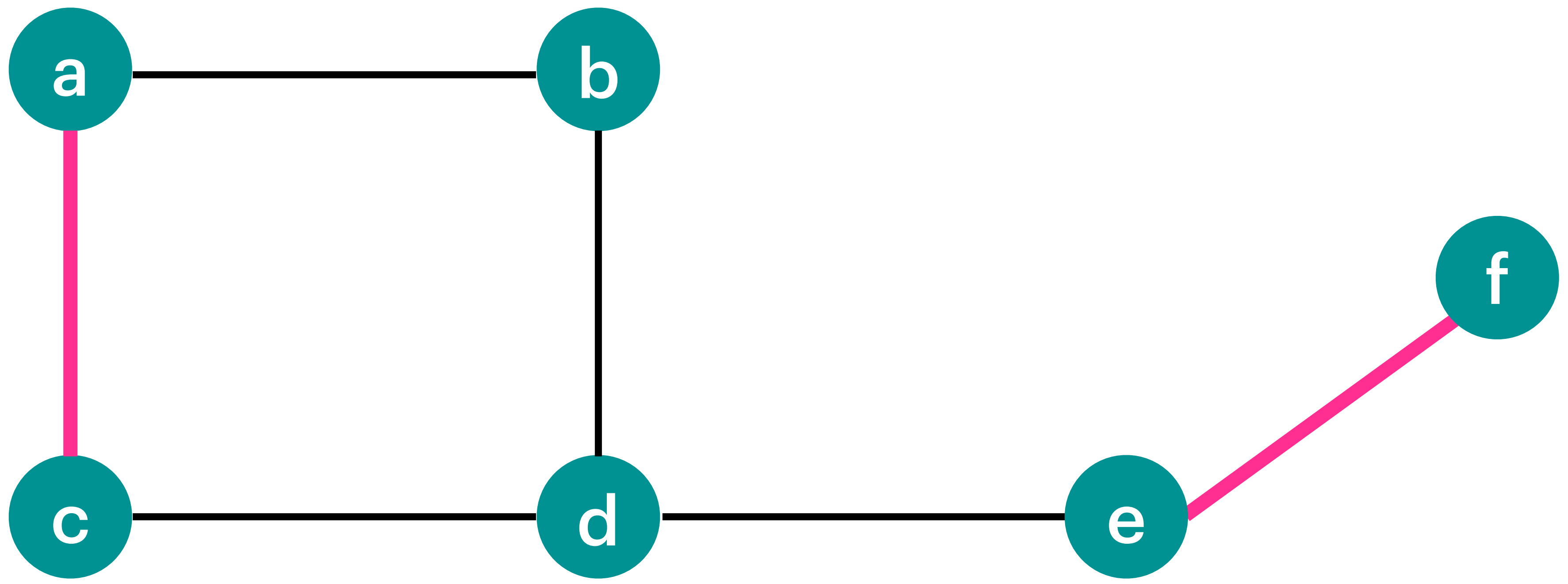


Matching
Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

Is this a matching ?



M_1 = {{a,c} , {e,f}}

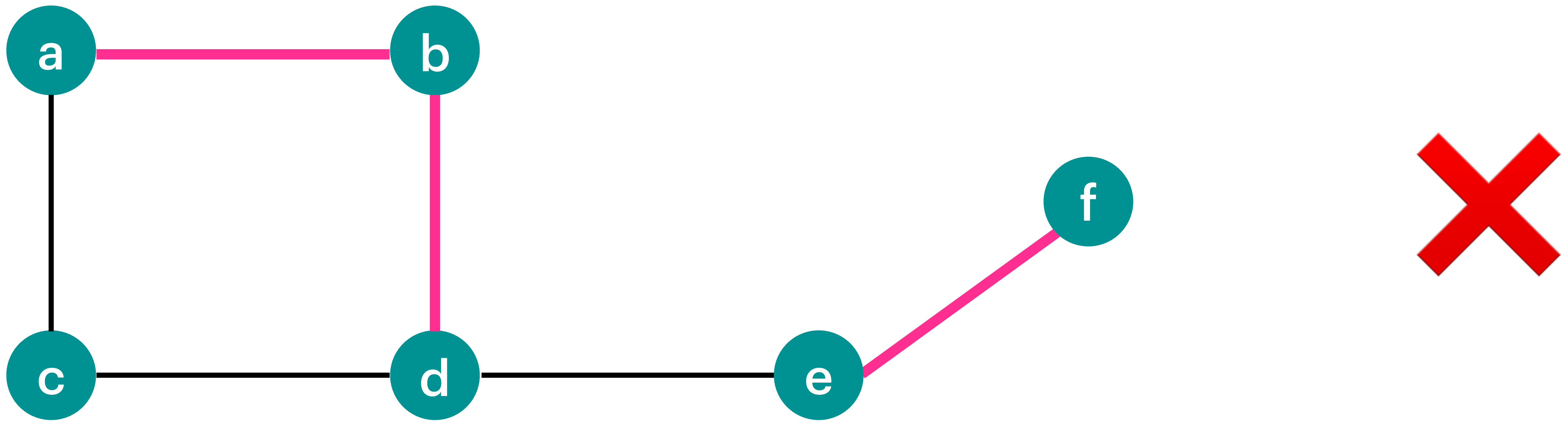
✅

Matching
Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

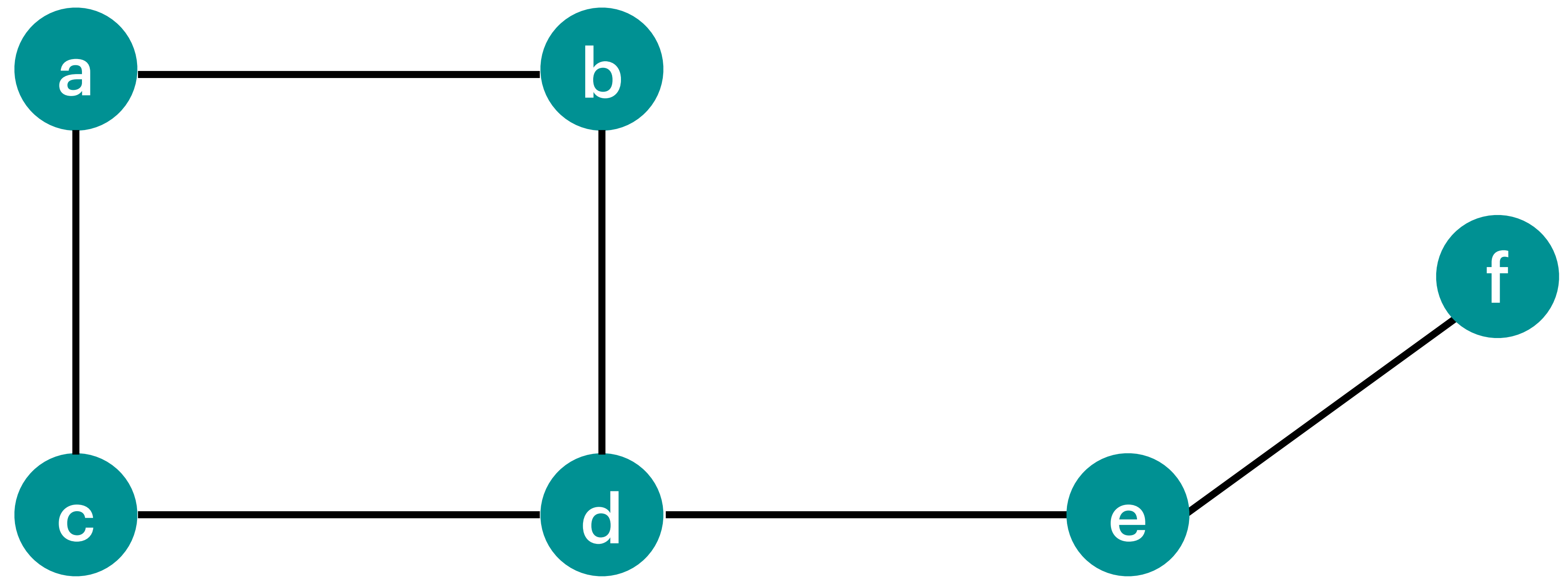
Is this a matching ?



Matching
Definitions

- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices



Matching
Definitions

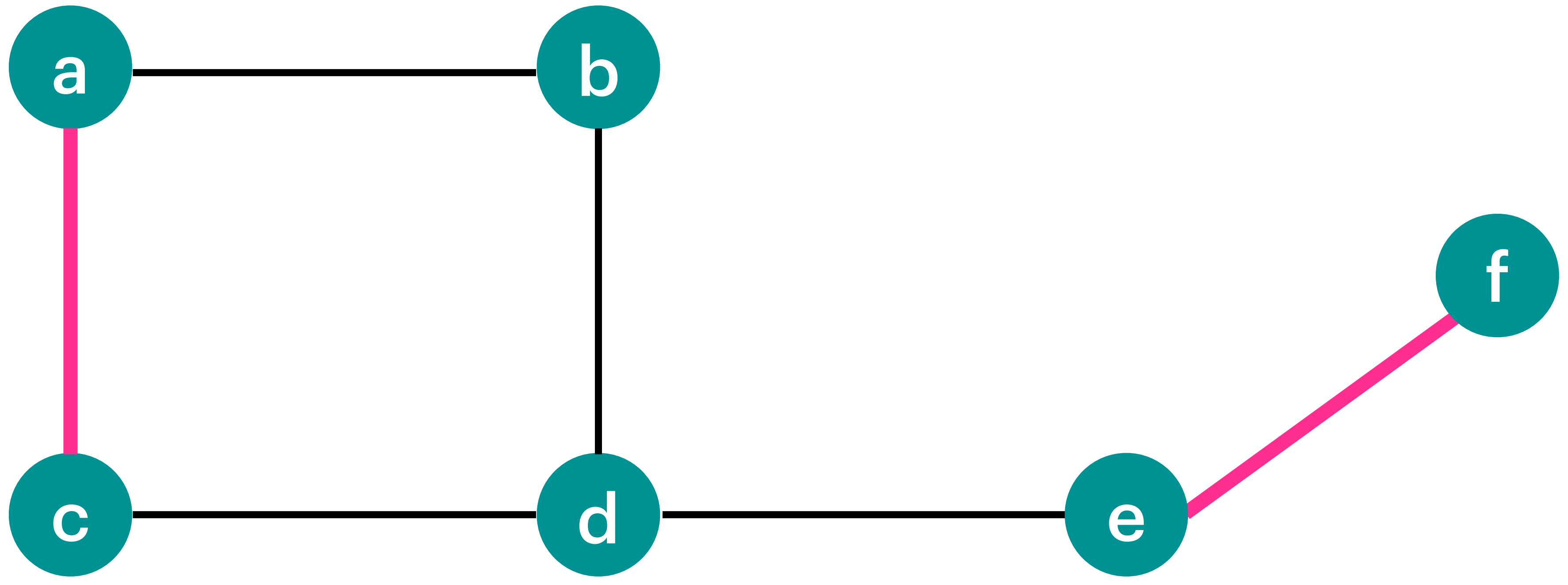
- Matching :
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

no two edges share common vertices

- covered (matched) :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

Matching
Definitions

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- covered (matched) :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v



Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M

- Perfect Matching :
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M
  - equivalently, if

$$M = \frac{|V|}{2}$$

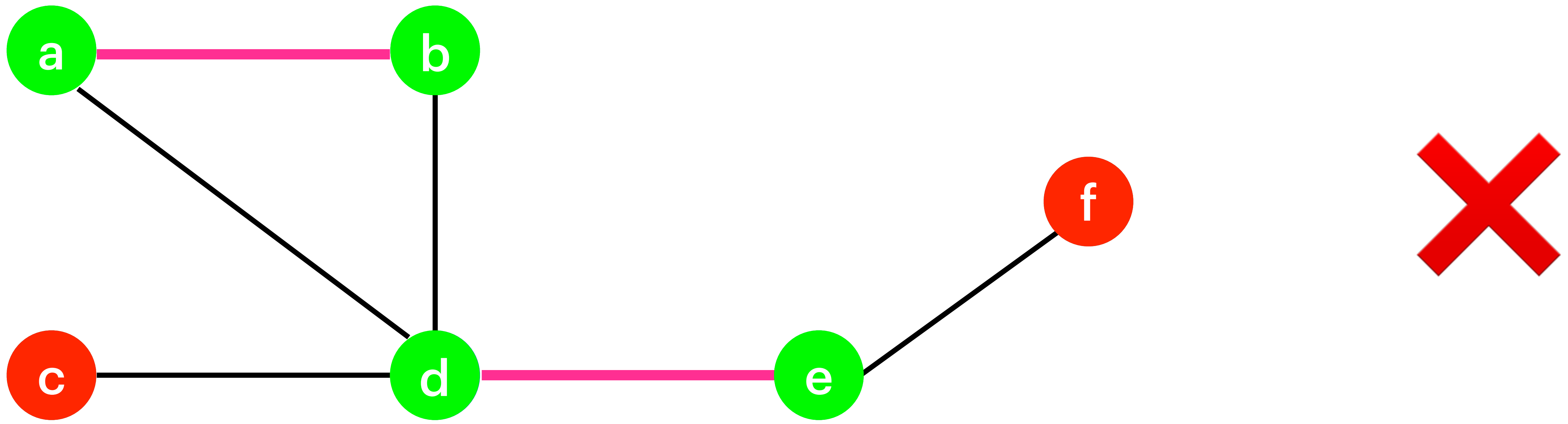
Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching : $M = \frac{|V|}{2}$
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M

Is this a perfect matching ?



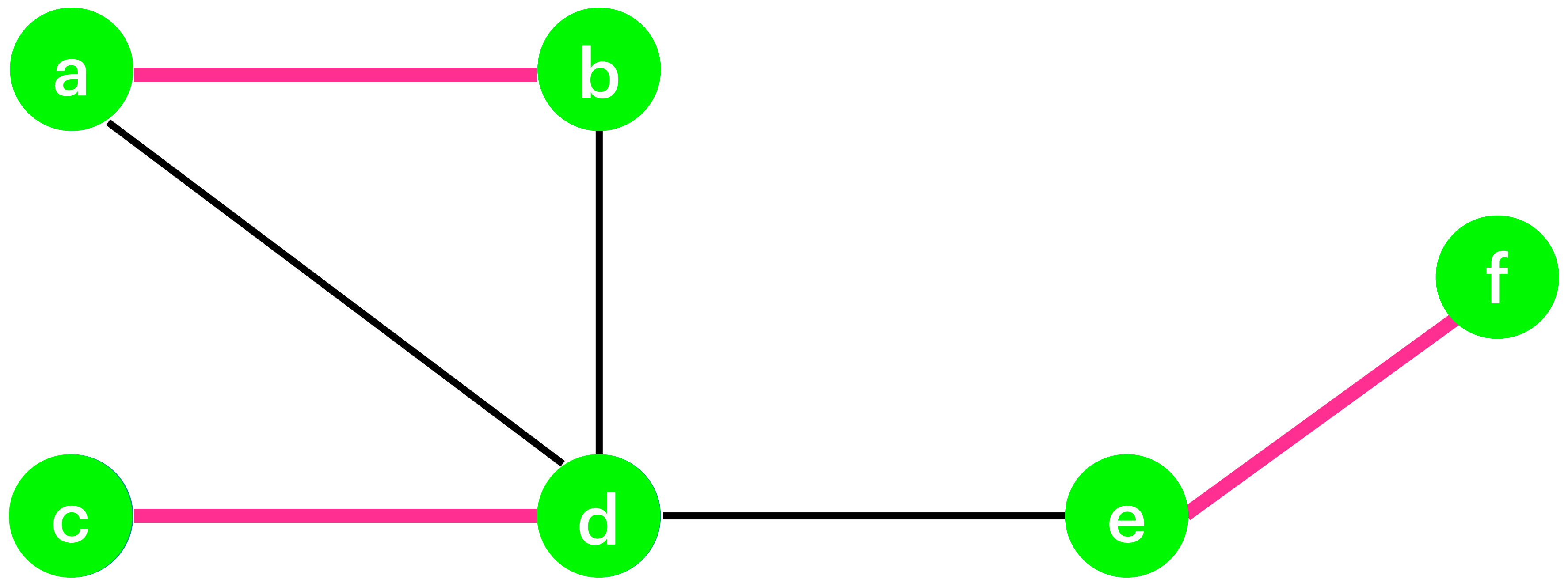
Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching : $M = \frac{|V|}{2}$
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M

Is this a perfect matching ?



✅

Matching

Definitions

- covered :
  - A vertex $v \subseteq V$ in a Graph $G = (V, E)$ is covered by M, if there exists an edge $e \in M$ that contains v

- Matching : no two edges share common vertices
  - A subset of edges $M \subseteq E$ in a Graph $G = (V, E)$ is called a Matching, if no vertex in the graph is incident to more than one edge from M
- Perfect Matching : $M = \frac{|V|}{2}$
  - A Matching M is called a Perfect Matching if every vertex is covered by exactly one edge from M

Is this a perfect matching ?

Matching

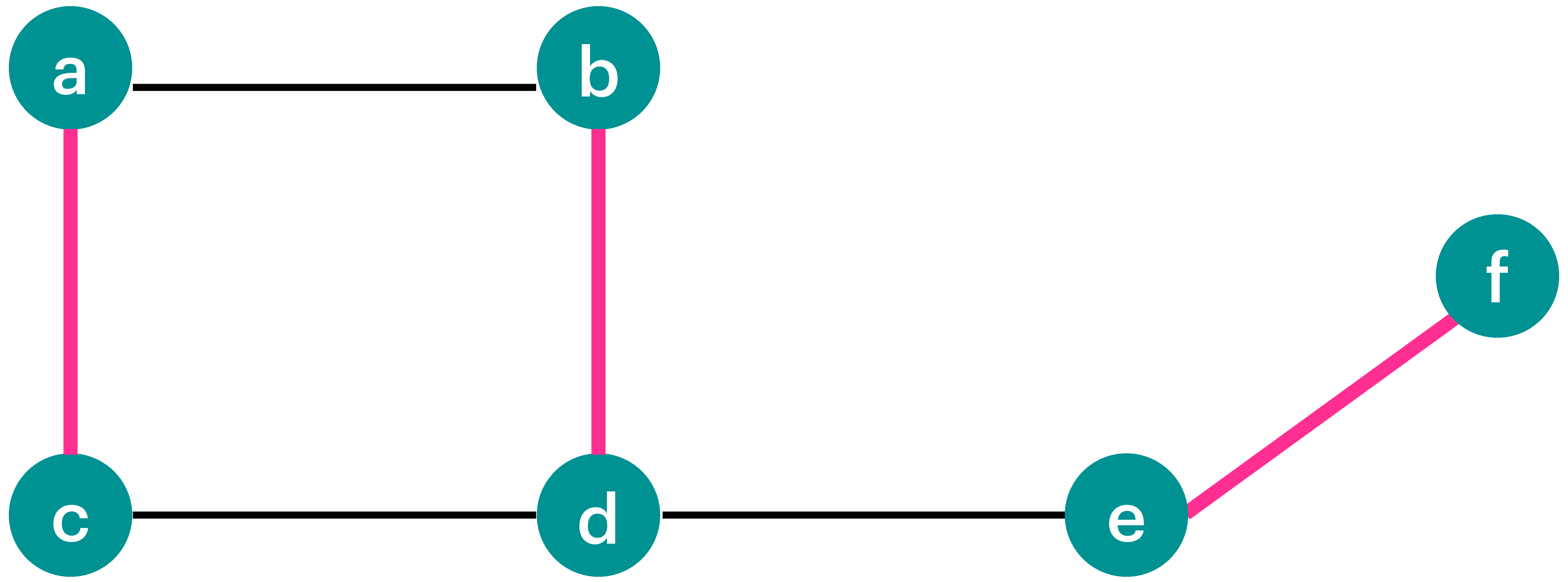
Definitions

- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subsetneq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Matching
Definitions

- inclusion-maximal : "no edge can be added to this matching"
  - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subsetneq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : "one can't find a bigger matching"
  - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Is this maximum ? ✅



Is this inclusion-maximal ? ✅

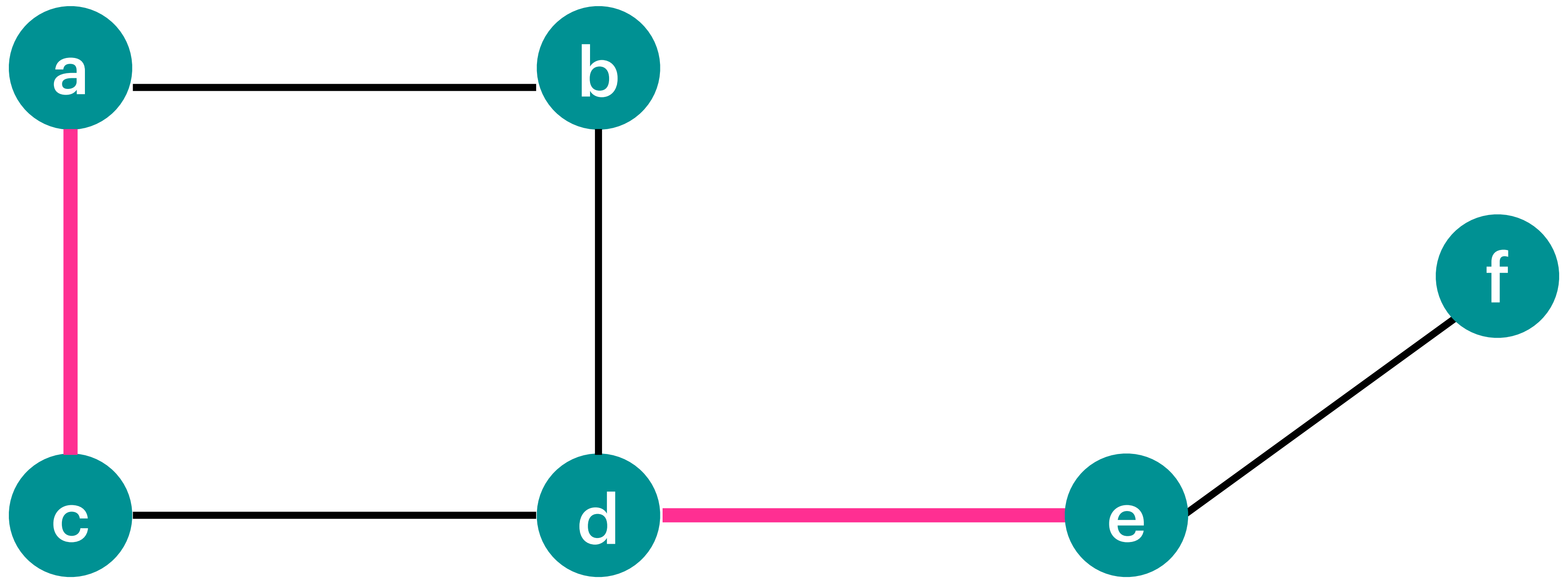
Matching

Definitions

- inclusion-maximal : “no edge can be added to this matching”
    - A matching $M \subseteq E$ is inclusion-maximal , if there is no other matching M' s.t. $M \subseteq M'$ (strict inclusion) and $|M'| > |M|$
- (cardinality-) maximum : “one can’t find a bigger matching”
    - A matching $M \subseteq E$ is (cardinality-) maximum , if there is no other matching M' s.t. $|M'| > |M|$

Is this maximum ?

❌



Is this inclusion-maximal ?

✅

Matching

Propositions

- M_{inc} : inclusion-maximal Matching , M_{max} : cardinality-maximum Matching

$$|M_{inc}| \leq |M_{max}|$$

$$|M_{inc}| \geq |M_{max}| / 2$$

Why ?

Every edge in M_{max} must have at least one endpoint in M_{inc}

Otherwise, that edge would be added to M_{inc}

$$|M_{max}| \leq |Endpoints\ in\ M_{inc}| = 2|M_{inc}|$$

Matching
Greedy Algorithm

- M_{inc} : inclusion-maximal Matching , M_{max} : cardinality-maximum Matching

$|M_{inc}| \leq |M_{max}|$

$|M_{inc}| \geq |M_{max}| / 2$

GREEDY-MATCHING (G)

1: $M \leftarrow \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: pick an arbitrary edge $e \in E$
4: $M \leftarrow M \cup \{e\}$
5: remove e and all incident edges in G

$|M_{Greedy}| \geq |M_{max}| / 2$

in $O(|E|)$

why ?

M_{Greedy} is inclusion-maximal

Matching

Augmenting ?

augment 1 of 2 *verb*

aug·ment ˈȯg-ˈment

augmented; augmenting; augments

Synonyms of *augment* >

transitive verb

1 : to make greater, more numerous, larger, or more intense

The impact of the report was *augmented* by its timing.

Synonyms & Similar Words

increase	expand	accelerate
boost	enhance	extend
raise	multiply	reinforce
amplify	intensify	maximize
strengthen	add (to)	enlarge
aggrandize	swell	escalate
stoke	compound	build up
supplement	pump up	up
heighten	complement	supersize
develop	prolong	lengthen
hype	inflate	skyrocket
dilate	distend	elongate

Matching

M - Augmenting Path

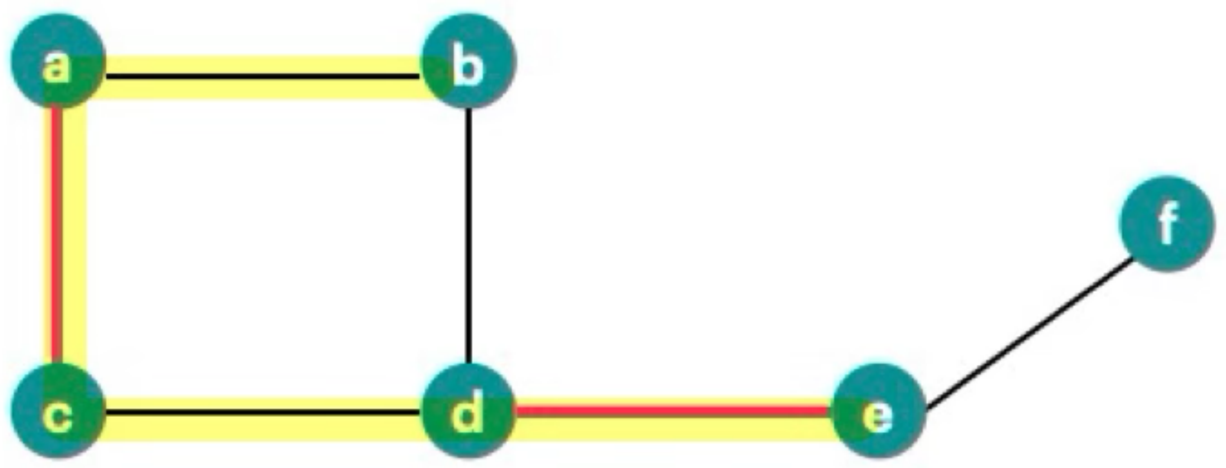
- Augmenting Path : “path with edges not in M, in M, … , not in M “
  - An augmenting path is an alternating path that starts from and ends on unmatched/not covered vertices
  - Alternating Path :
    - An alternating path is a path that begins with an unmatched/not covered vertex whose edges belong alternately to the matching and not to the matching

Matching

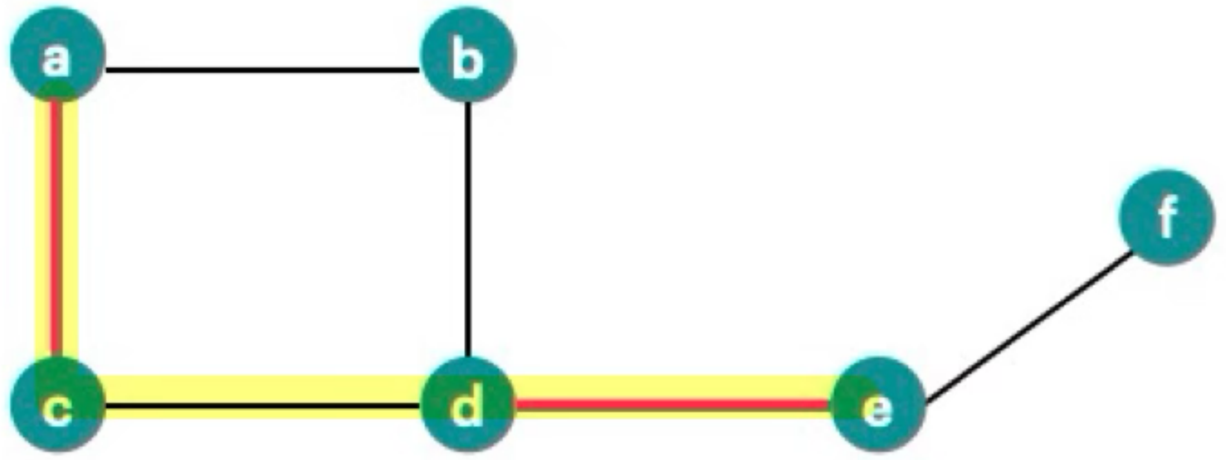
M - Augmenting Path

Is this an augmenting path ?

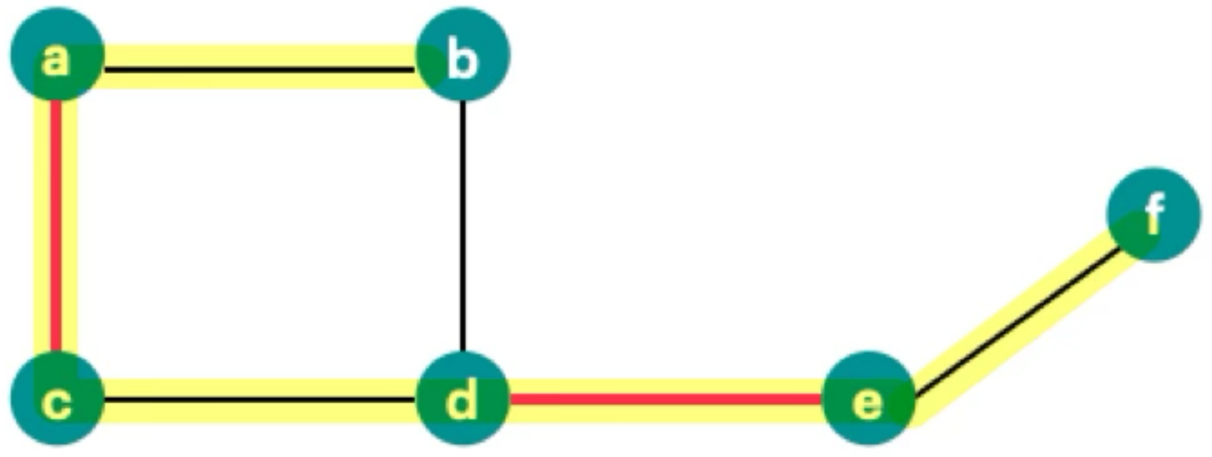
- Augmenting Path : "path with edges not in M, in M, … , not in M "
  - An augmenting path is an alternating path that starts from and ends on unmatched/not covered vertices
  - Alternating Path :
    - An alternating path is a path that begins with an unmatched/not covered vertex whose edges belong alternately to the matching and not to the matching



— Matching M

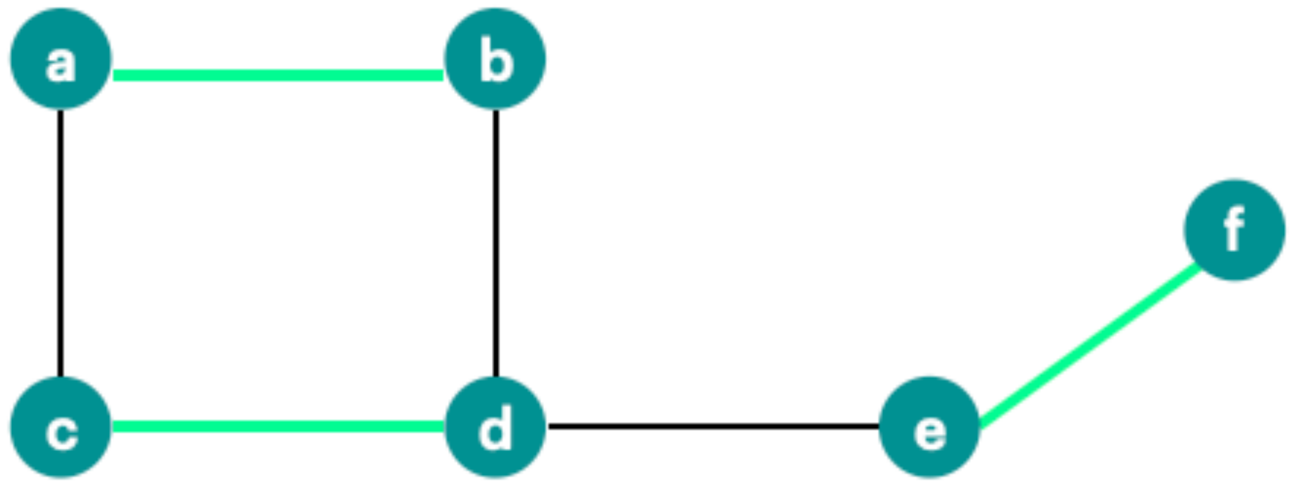


— Matching M



— Matching M

Idea : By swapping along M we can improve the matching



— Matching M'

Matching

Swapping ?

$$A \oplus B$$

Elements that are in A or in B but not in both

$$A = \{1,2,3\}$$

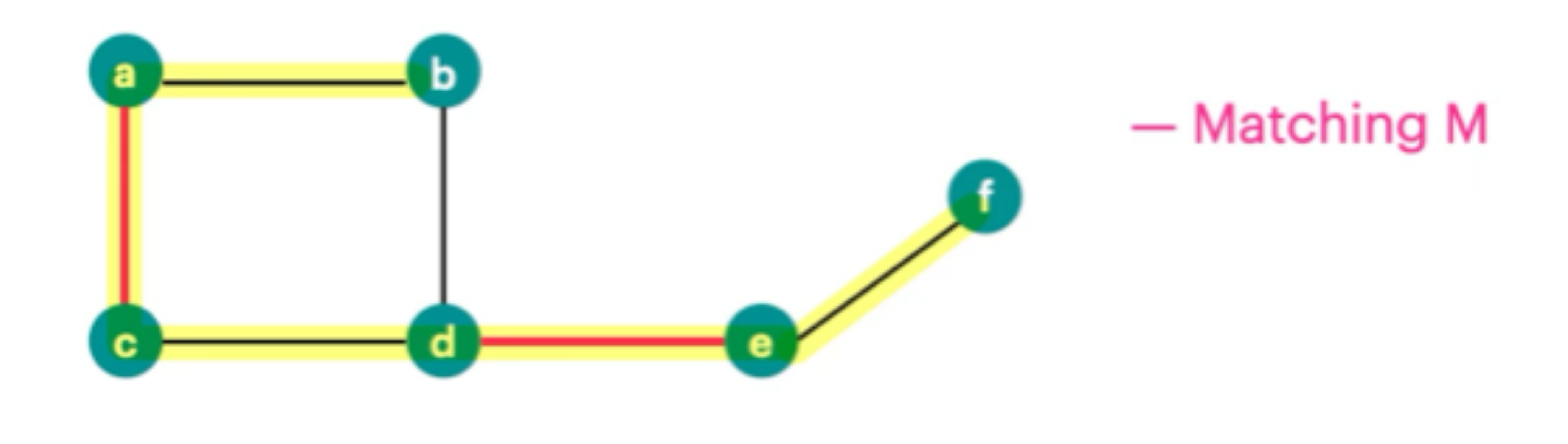
$$B = \{3,4,5\}$$

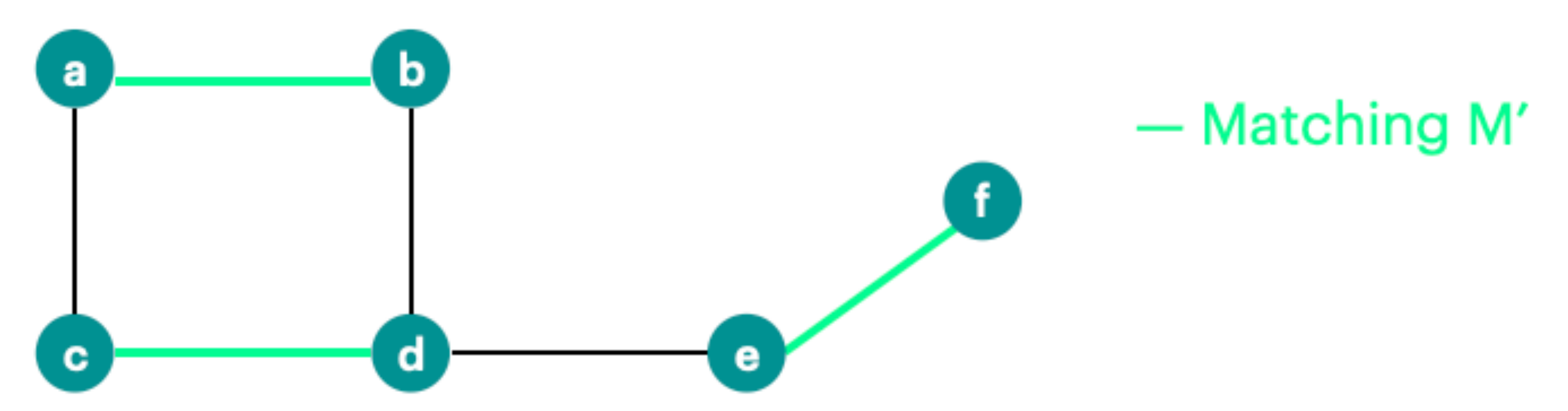
$$A \oplus B = \{1,2,4,5\}$$

Matching

Swapping ?

$$M' := M \oplus P$$





— M-augmenting path P

Matching

Berge's Theorem

- Augmenting Path : "path with edges not in M, in M, ... , not in M "
  - An augmenting path is an alternating path that starts from and ends on unmatched/not covered vertices
  - Alternating Path :
    - An alternating path is a path that begins with an unmatched/not covered vertex whose edges belong alternately to the matching and not to the matching

A Matching M is (cardinality-) maximum $\Longleftrightarrow$ There's no M-augmenting path

Idea : To find the maximum matching, update/improve the matching until there is no augmenting path left

Matching
Algorithm

💡 Idea : Update/improve the matching until there is no augmenting path left

Input : $G = (V, E)$

Output : maximum matching M

Algorithm :

Start with $M = \varnothing$

while ∃ augmenting path P

$$M = M \oplus P$$

return M

How do we find the augmenting path P ?

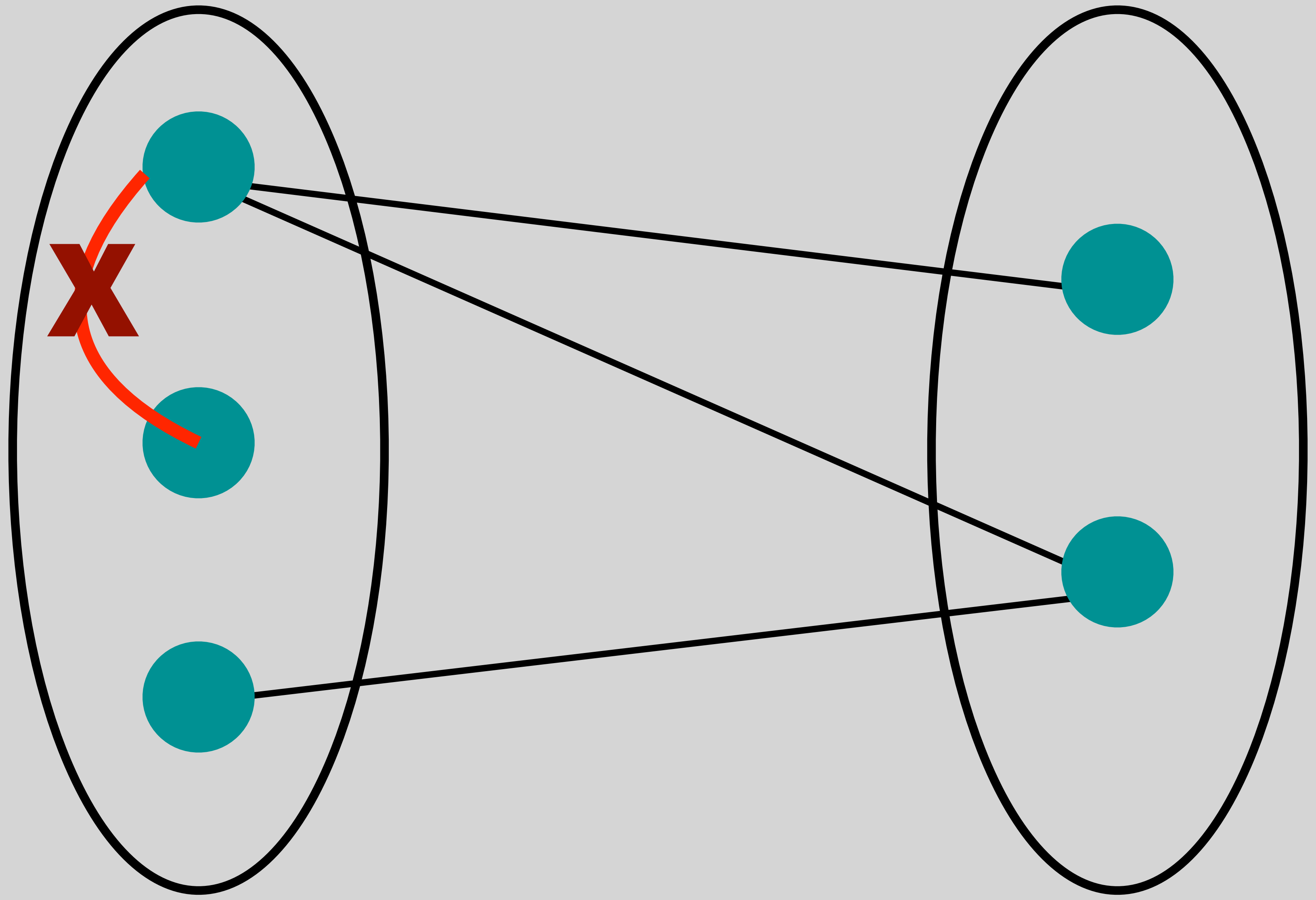
bipartite Gs : with BFS

general Gs in $O(|V||E|)$

Matching
Definitions

- Bipartite Graph :
  - A graph G is bipartite , if you can split the set of vertices V into two sets U, V s.t. :

$$E \subseteq \{ \{u, v\} \ : u \in U, v \in V\}$$



Matching

Definitions

- k-regular
  - A graph G is k-regular , if every vertex has a degree of k

$$deg(v) = k \quad \forall v \in V$$

Matching

Perfect Matching finding

bipartite ?	k-regular	runtime
✅	2^K	O(|E|)
✅	k	O(|E|)
✅	-	O(|V| · |E|)

Matching

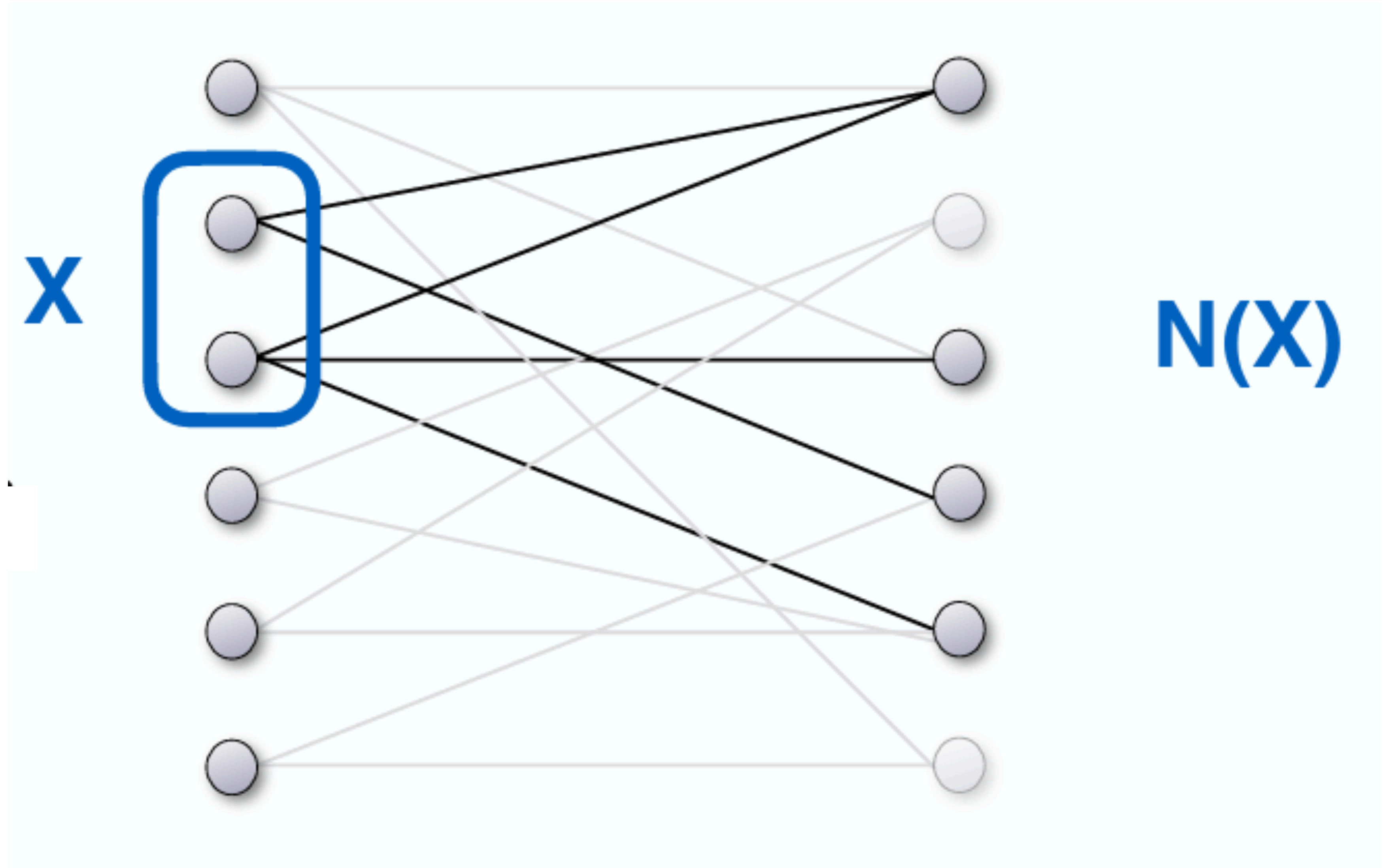
Hall's Marriage Theorem

A bipartite $G = (A \cup B, E)$

has a Matching M with cardinality $|M| = |A|$

$$\iff$$

$$\forall X \subseteq A : |X| \leq |N(X)|$$



N(X) := "neighbours of vertices in X"

Corollary : Every k-regular bipartite G has a perfect matching

Matching

Algorithm - revisit

💡 Idea : Update/improve the matching until there is no augmenting path left

Input : $G = (V, E)$

Output : maximum matching M

Algorithm :

Start with $M = \varnothing$

while $\exists$ augmenting path P

$M = M \oplus P$

return M

How do we find the augmenting path P ?

bipartite Gs : with BFS

general Gs in $O(|V||E|)$

BFS + this -> O (|V| |E|)

Matching

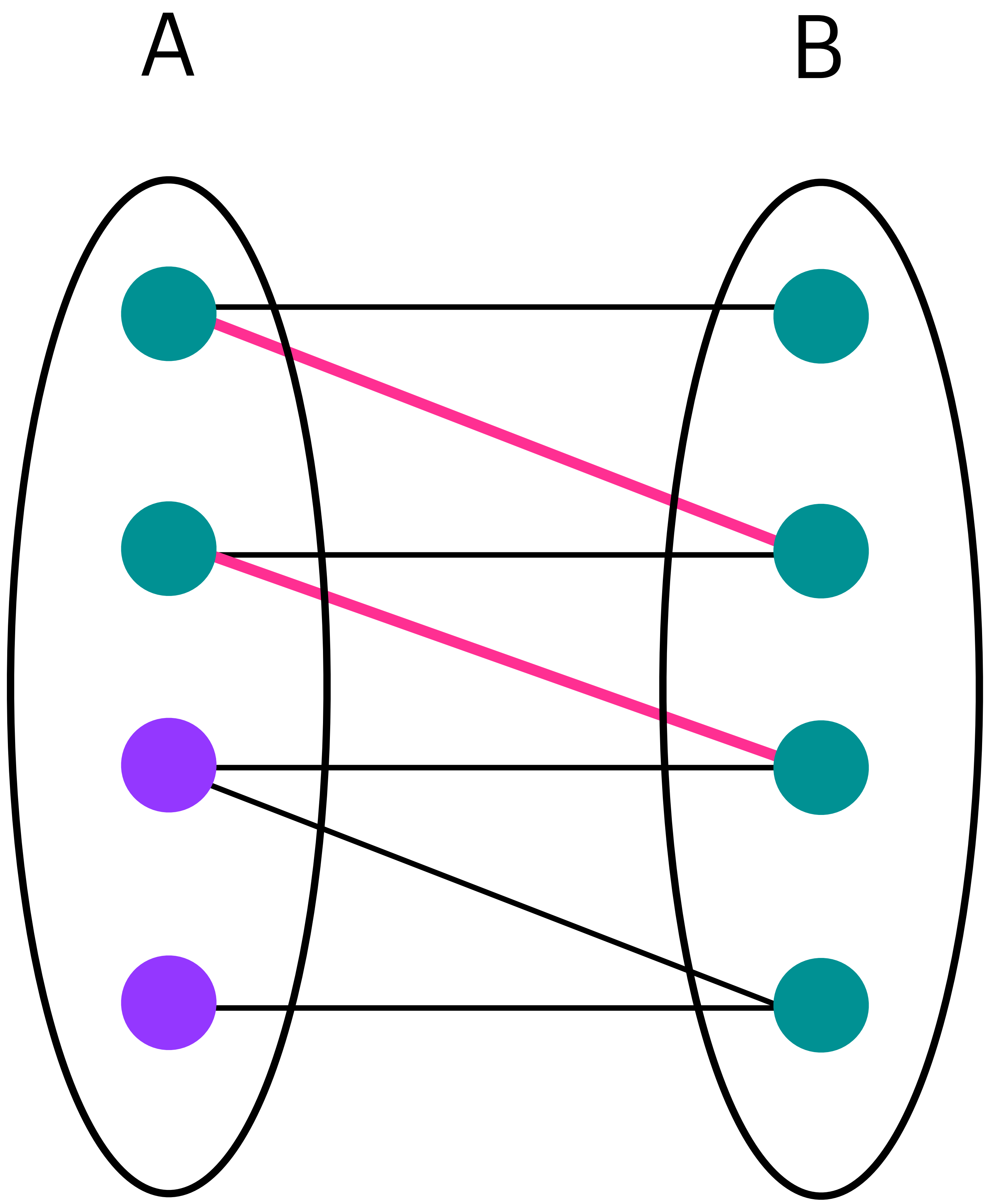
BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}



Matching

BFS for augmenting paths

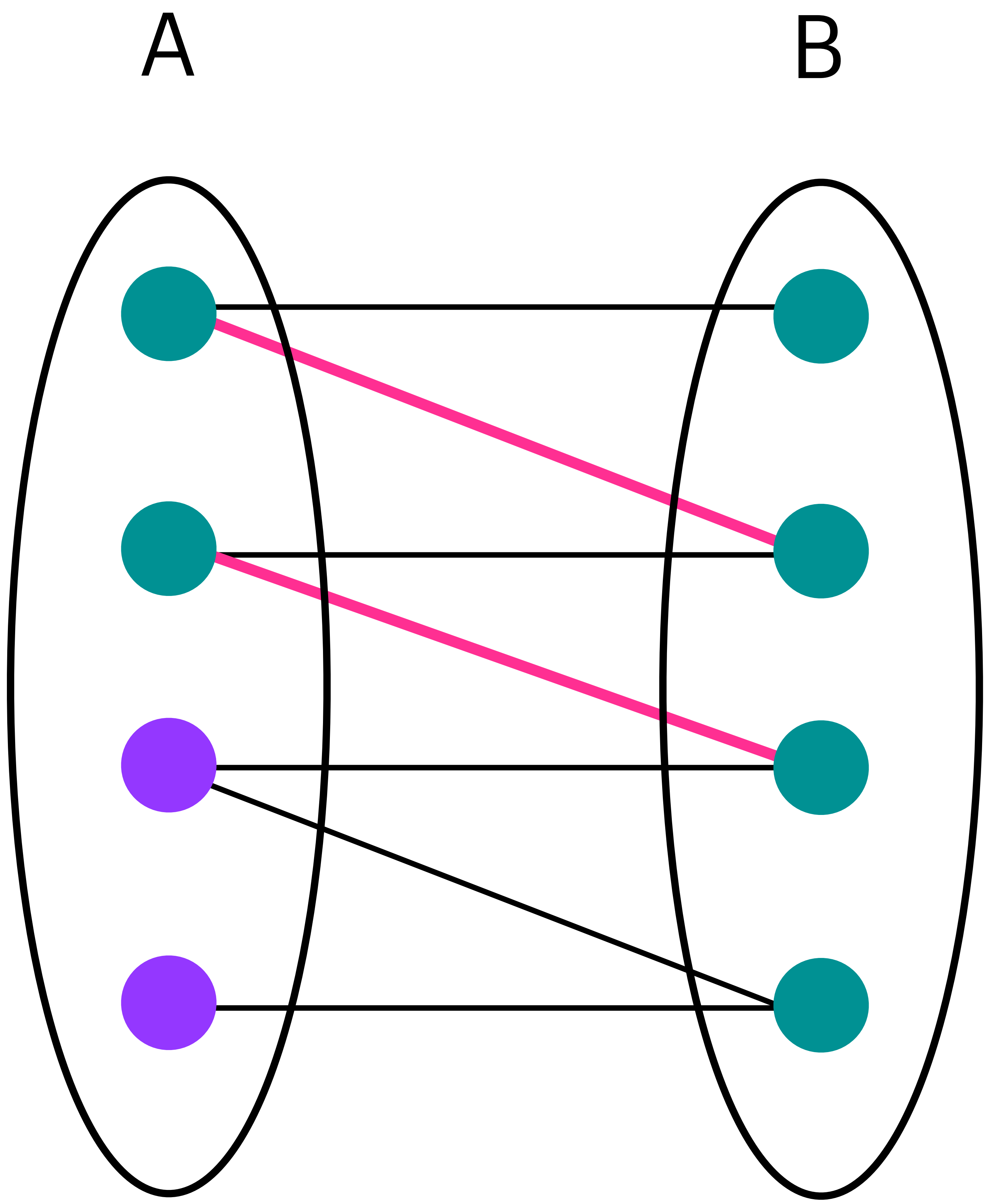
Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited



Matching

BFS for augmenting paths

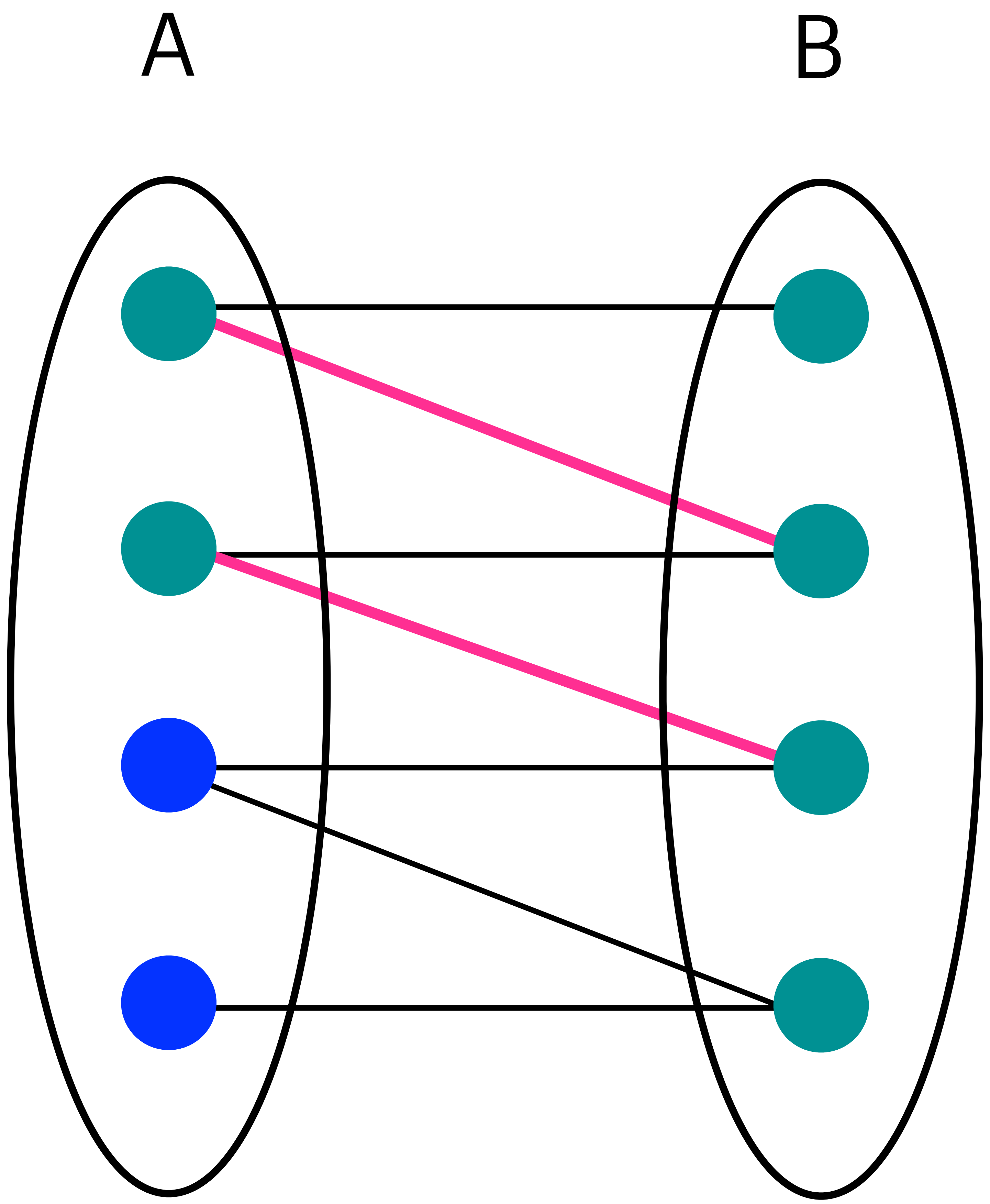
Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

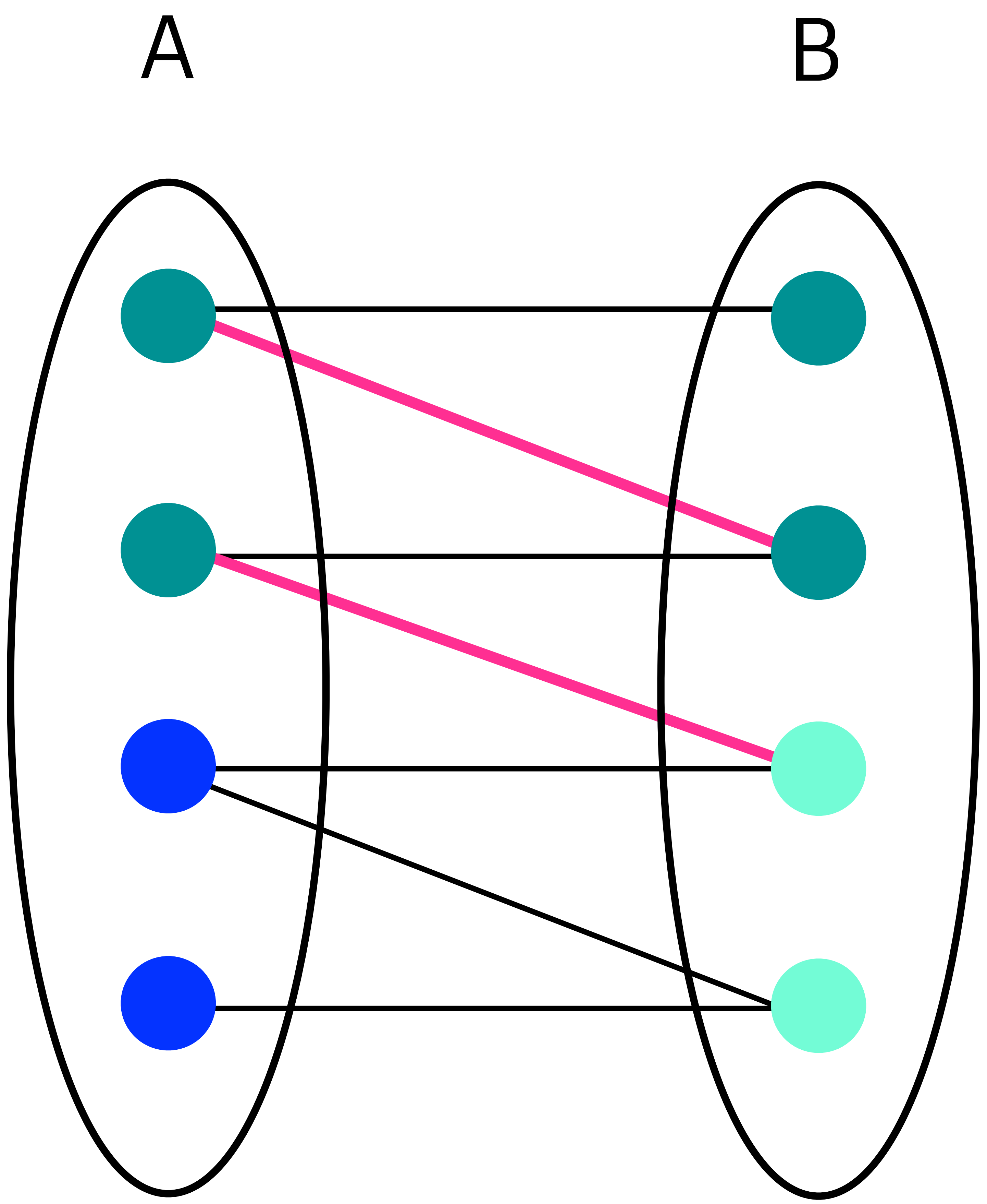
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)

i = 1



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

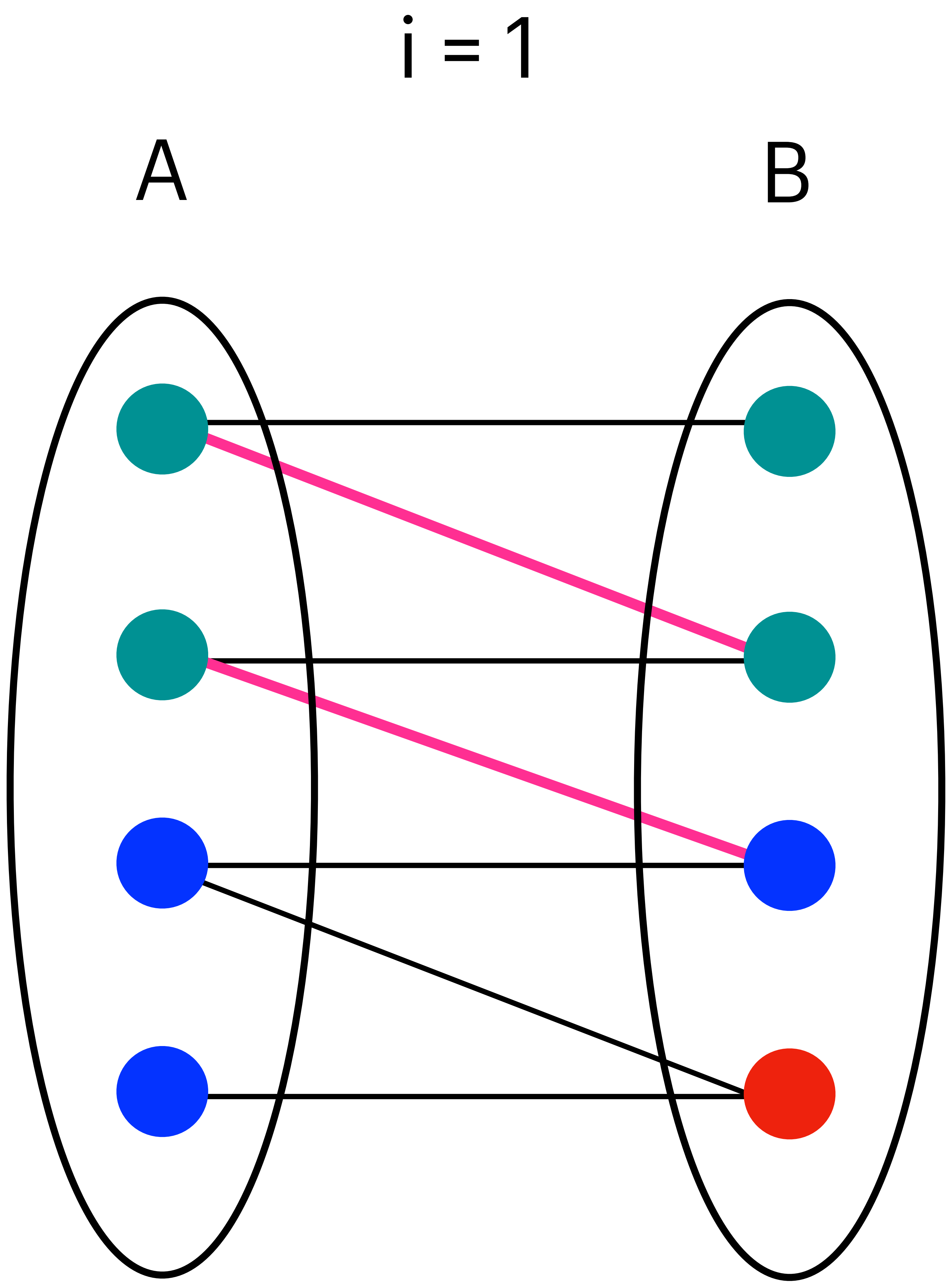
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

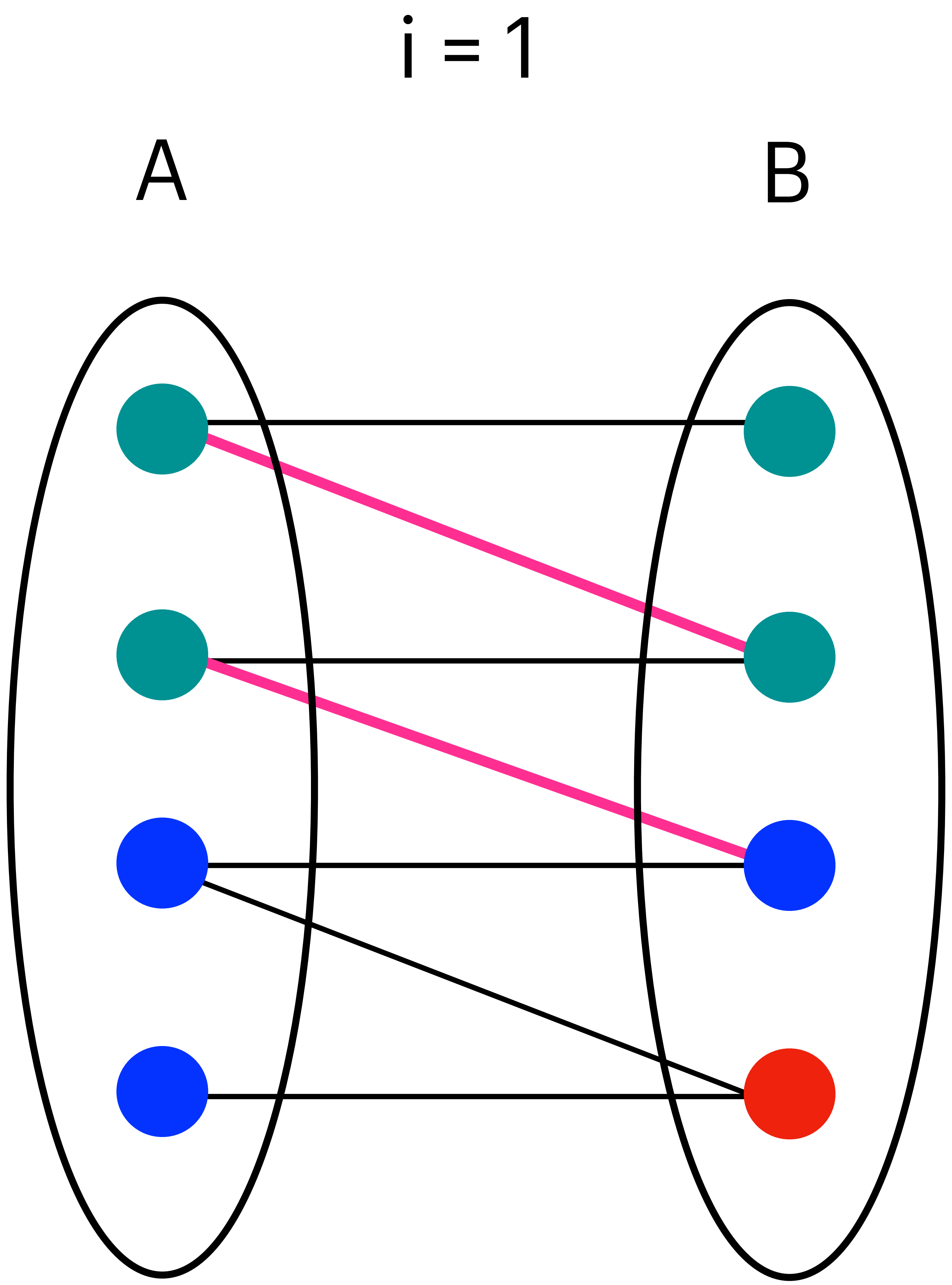
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

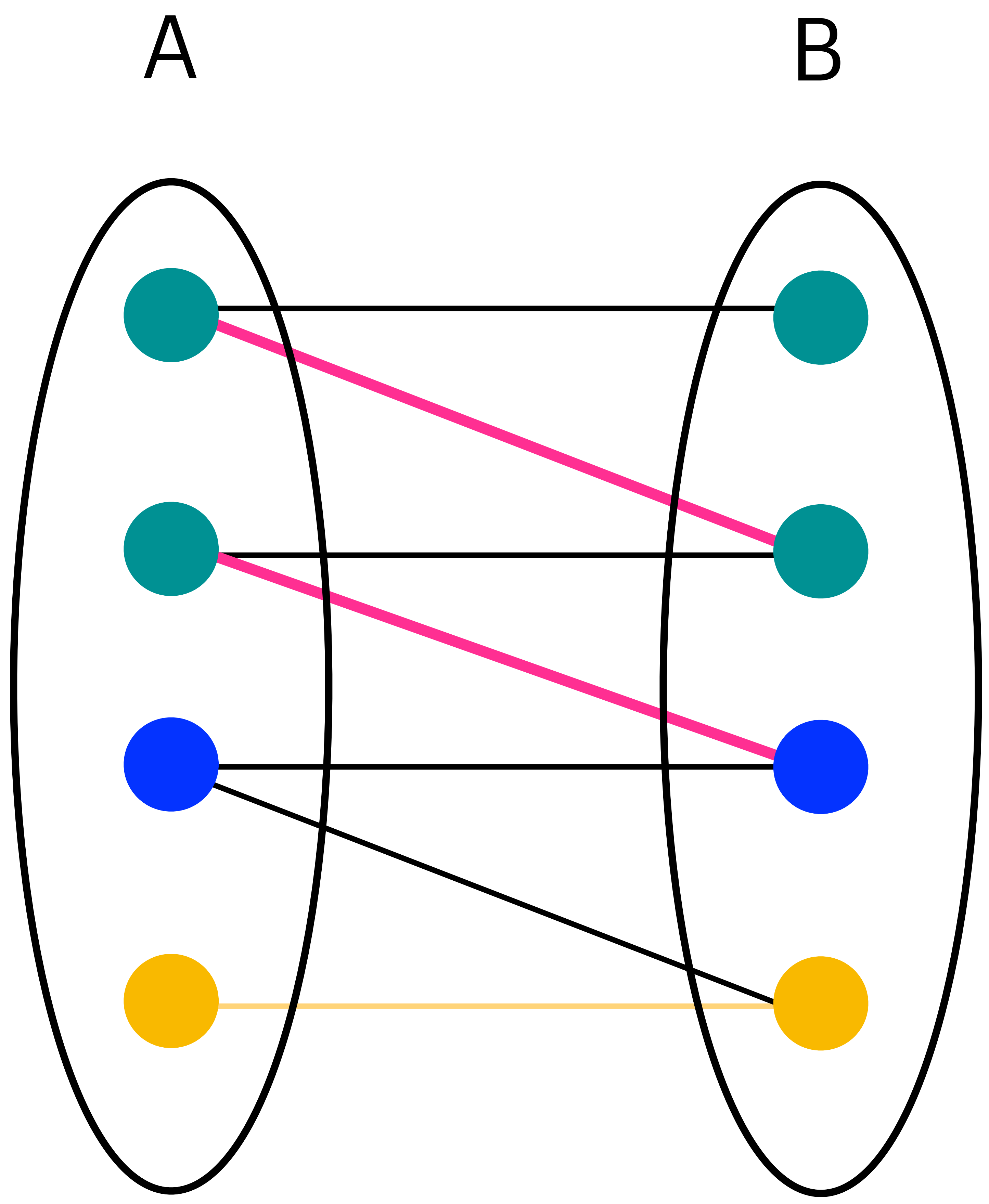
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking)

i = 1



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

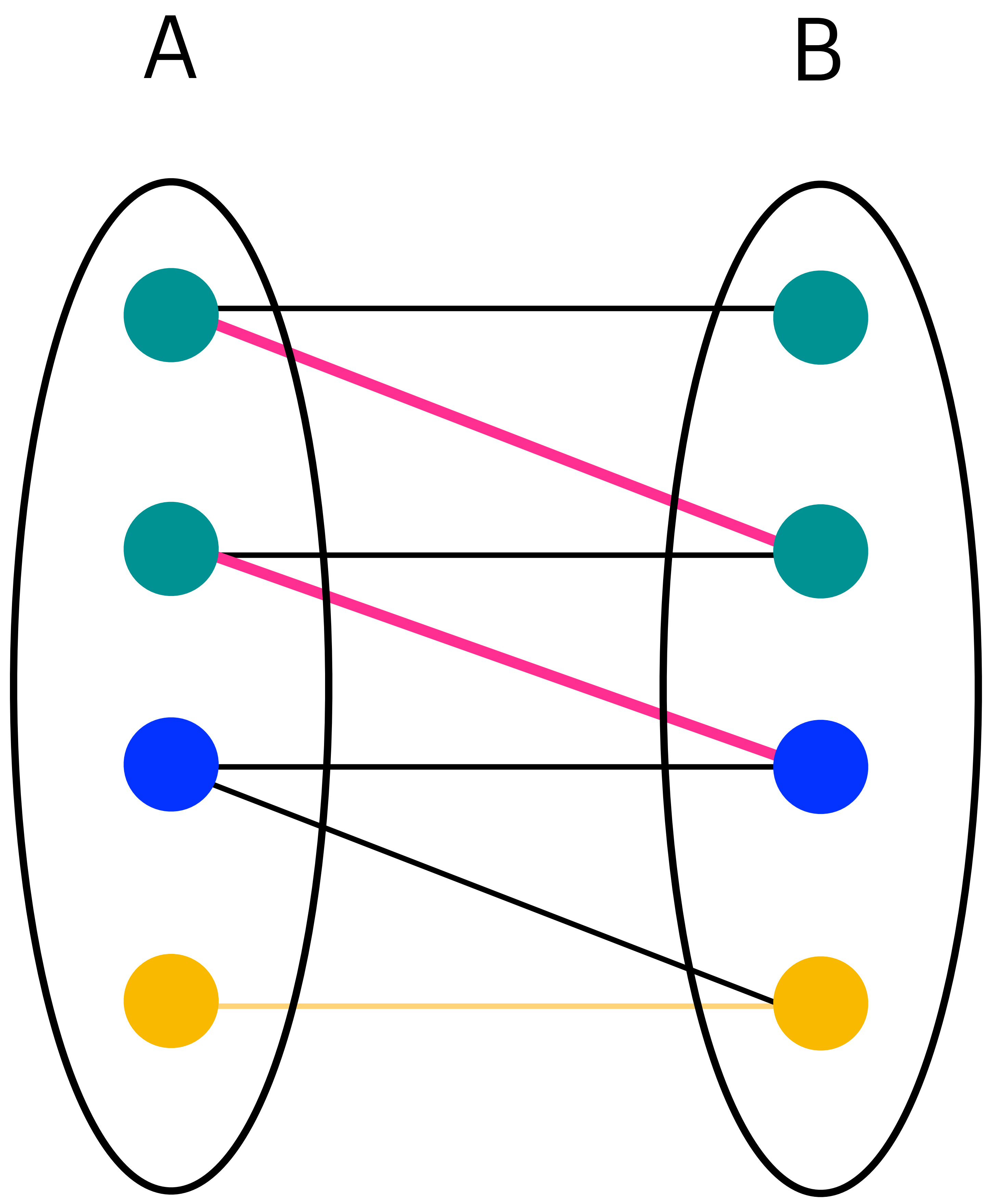
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 1



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

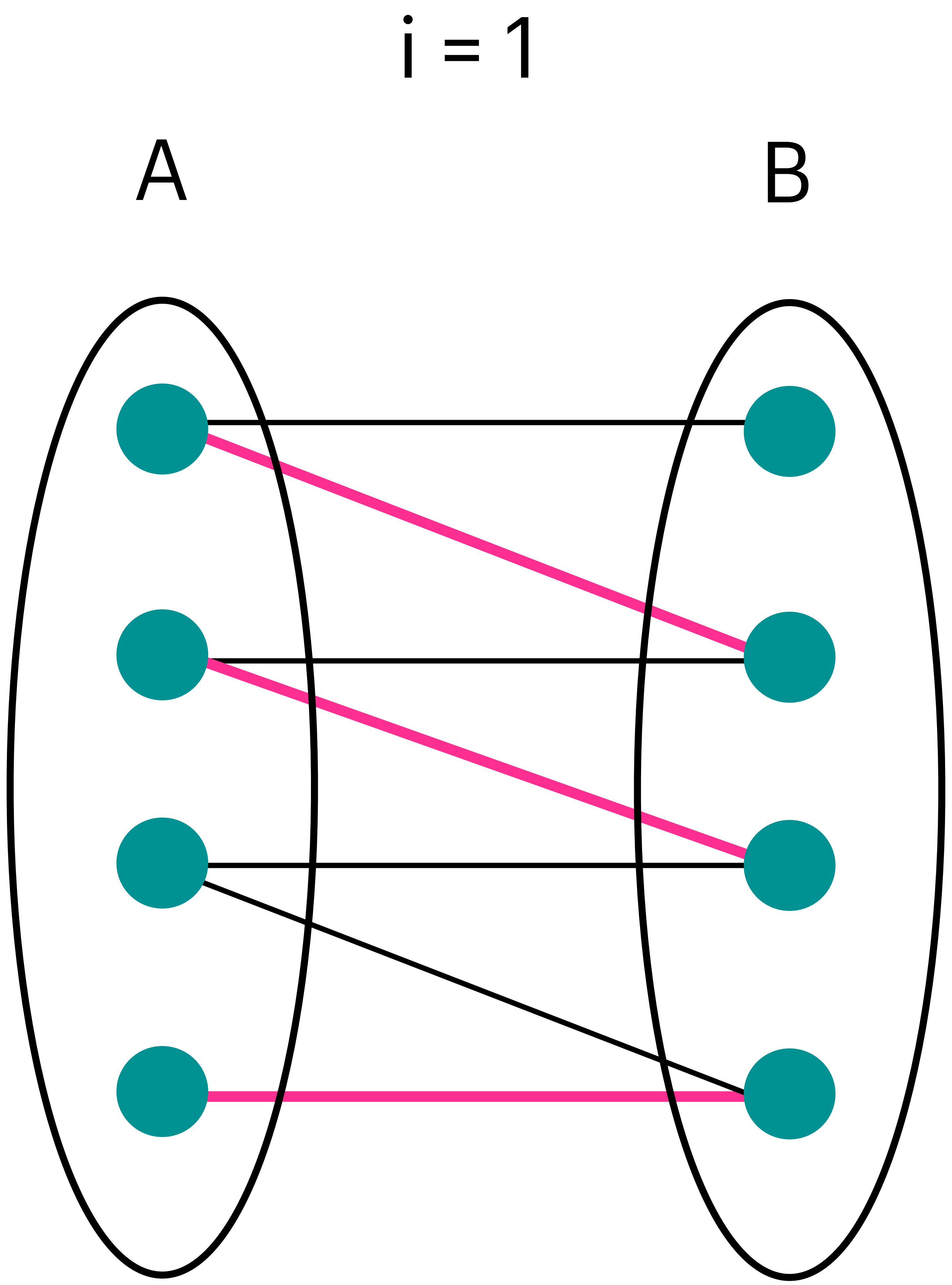
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

RESTART

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

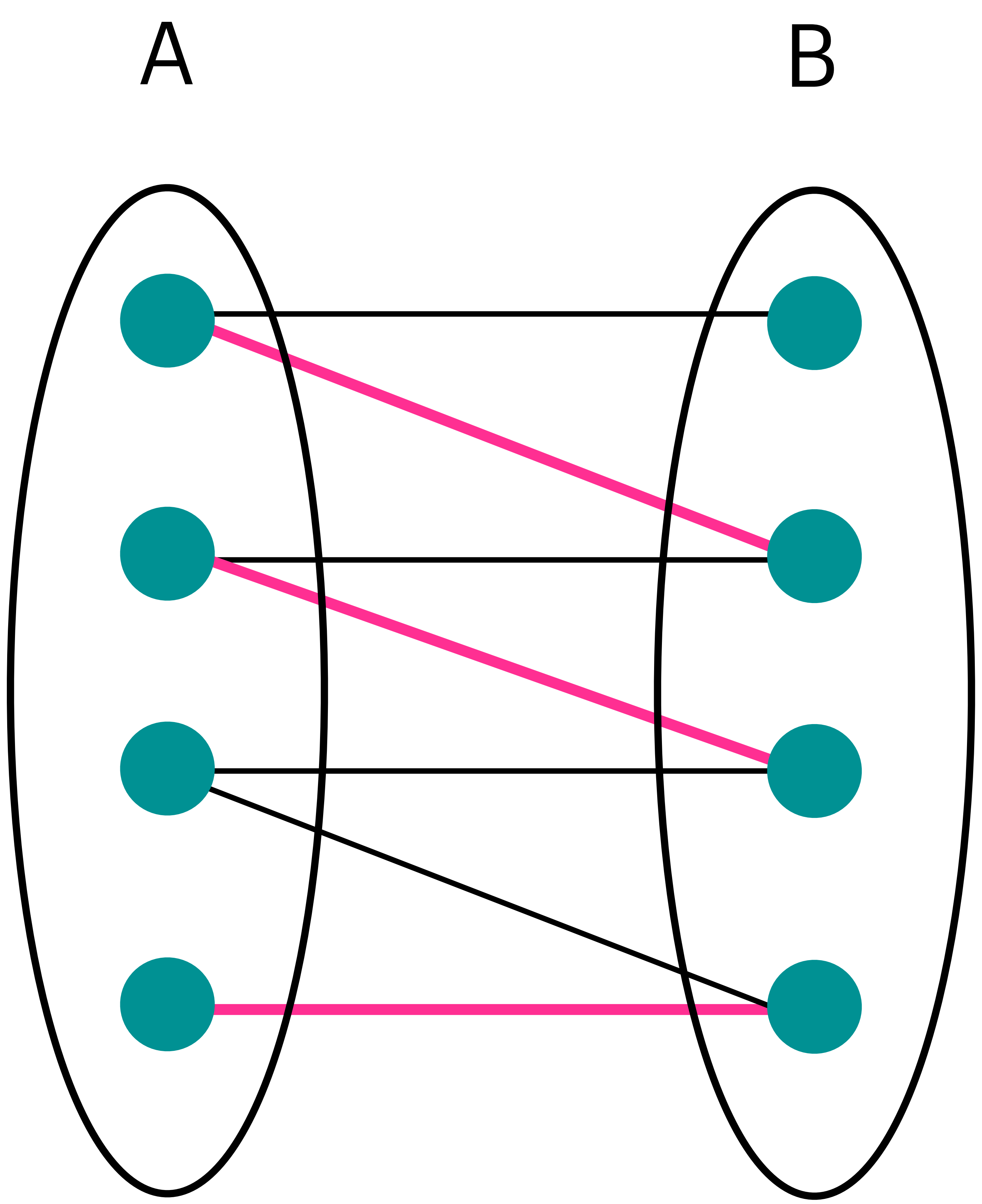
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 1



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

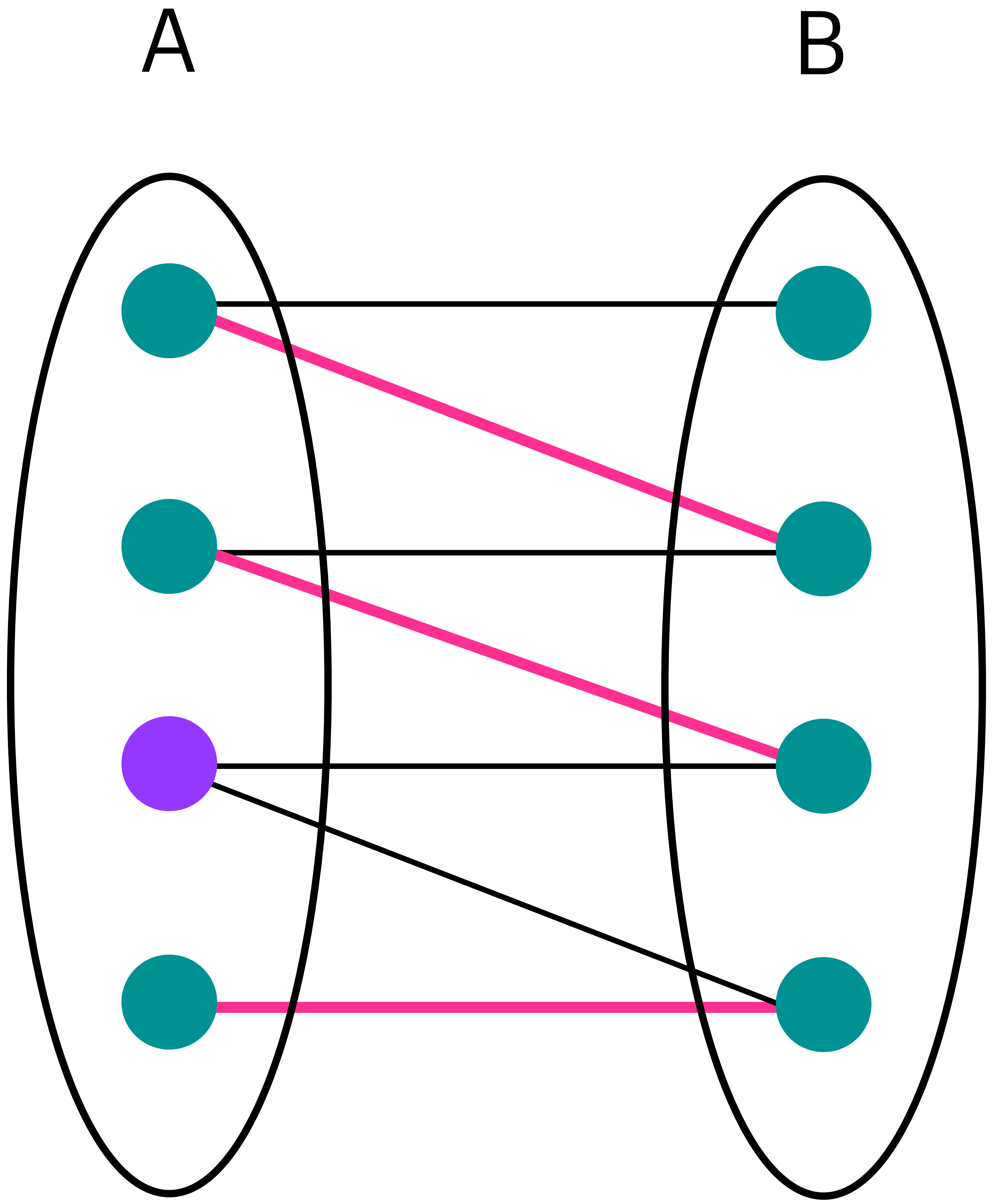
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

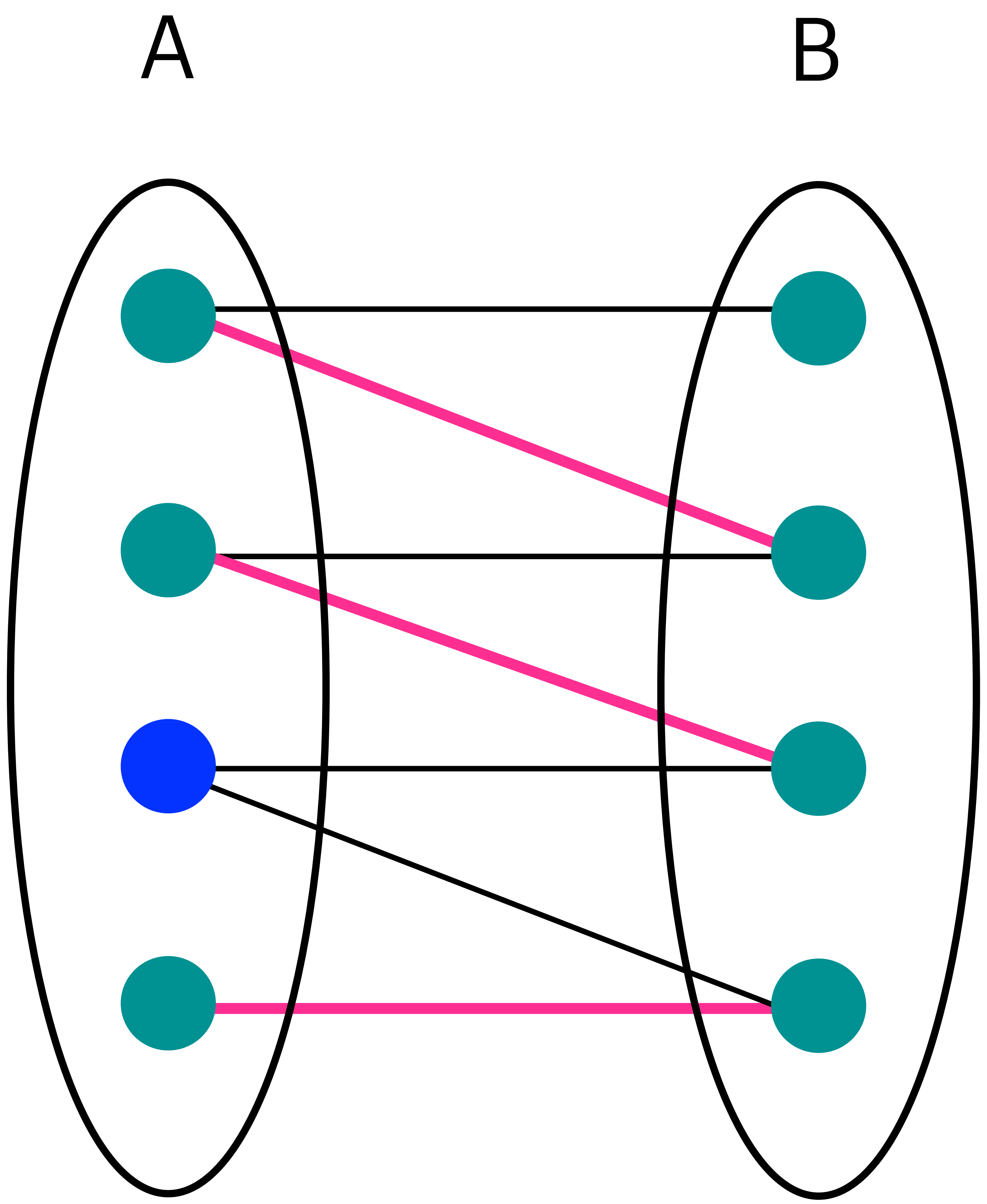
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

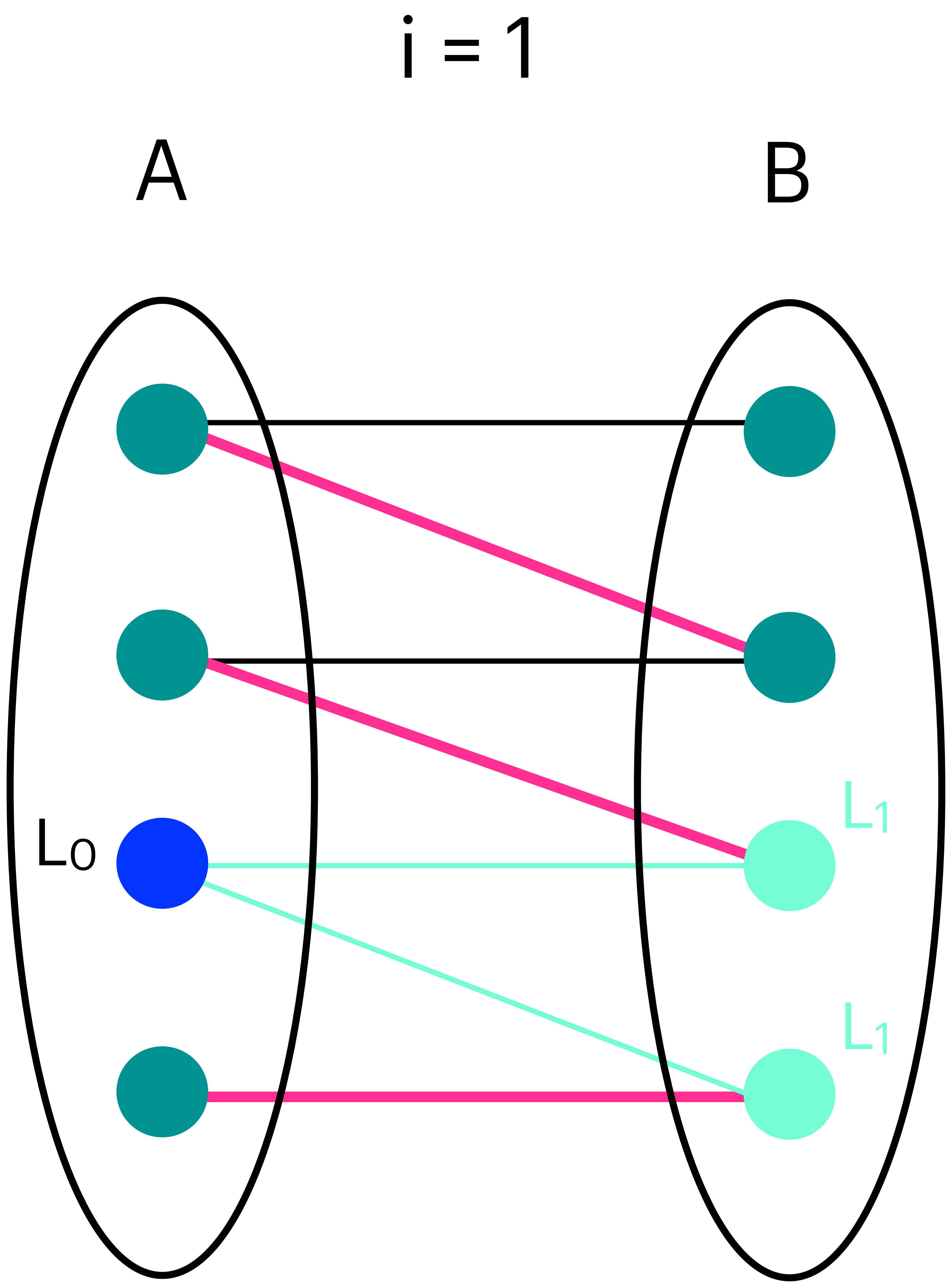
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

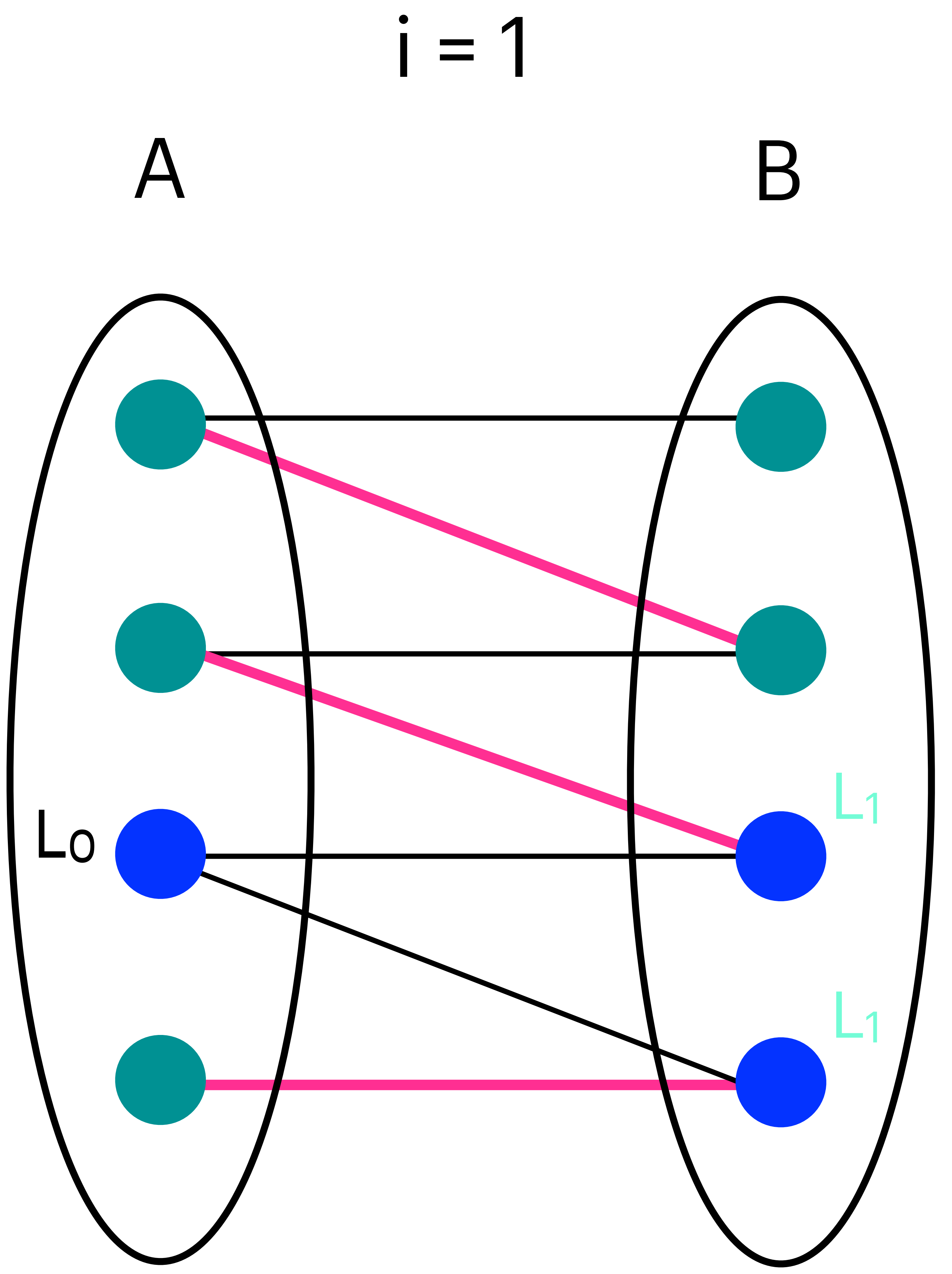
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

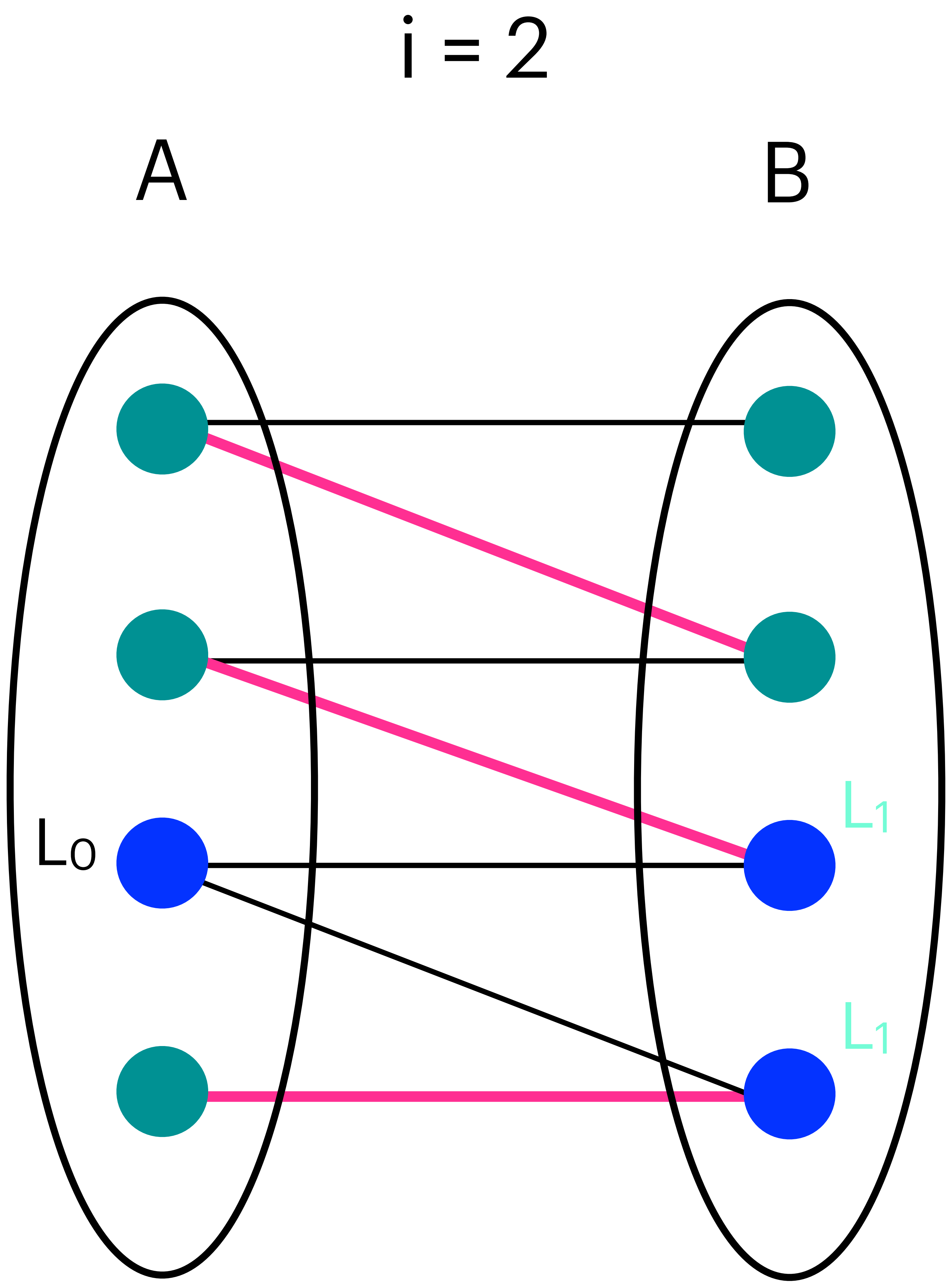
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

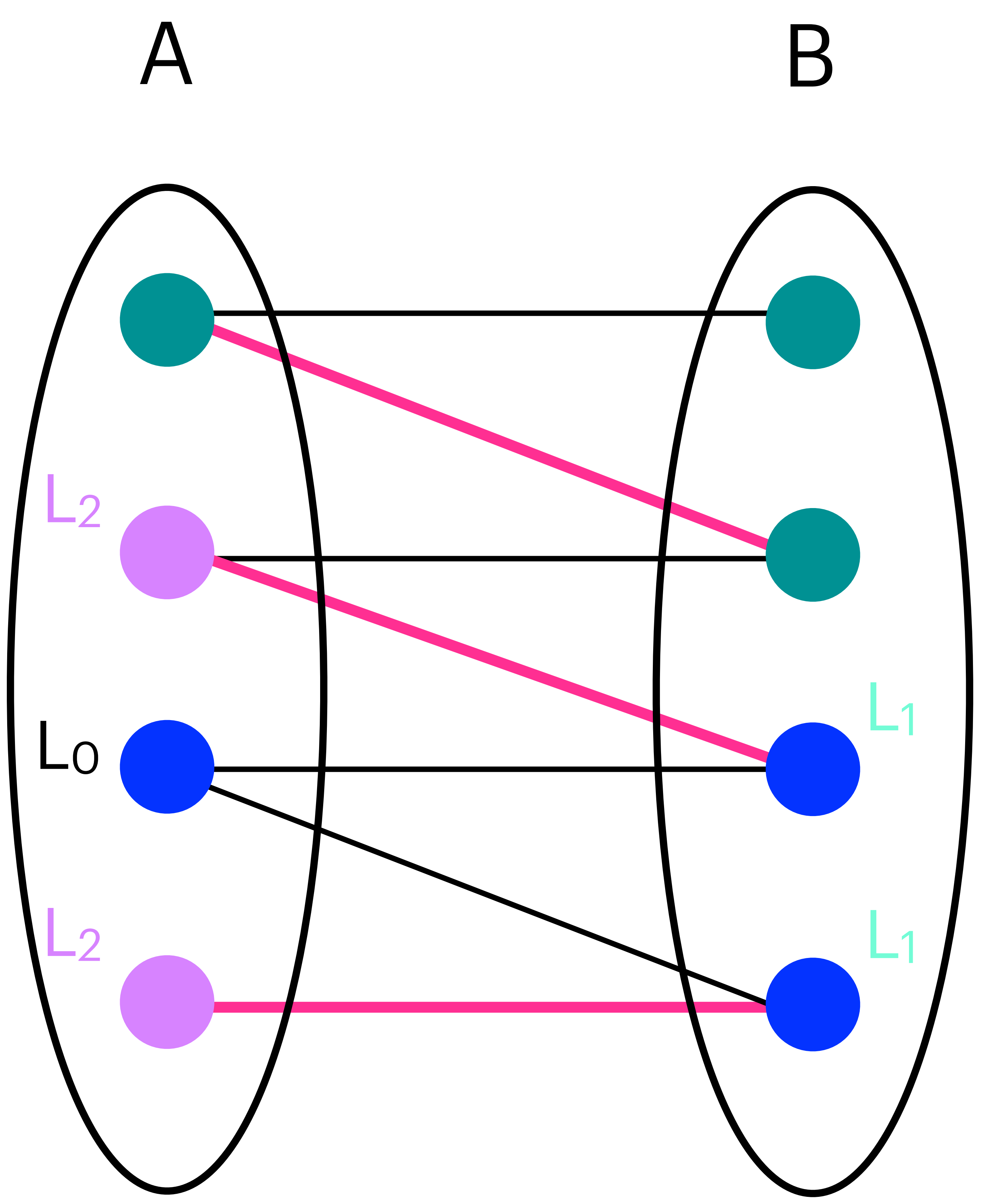
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 2



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

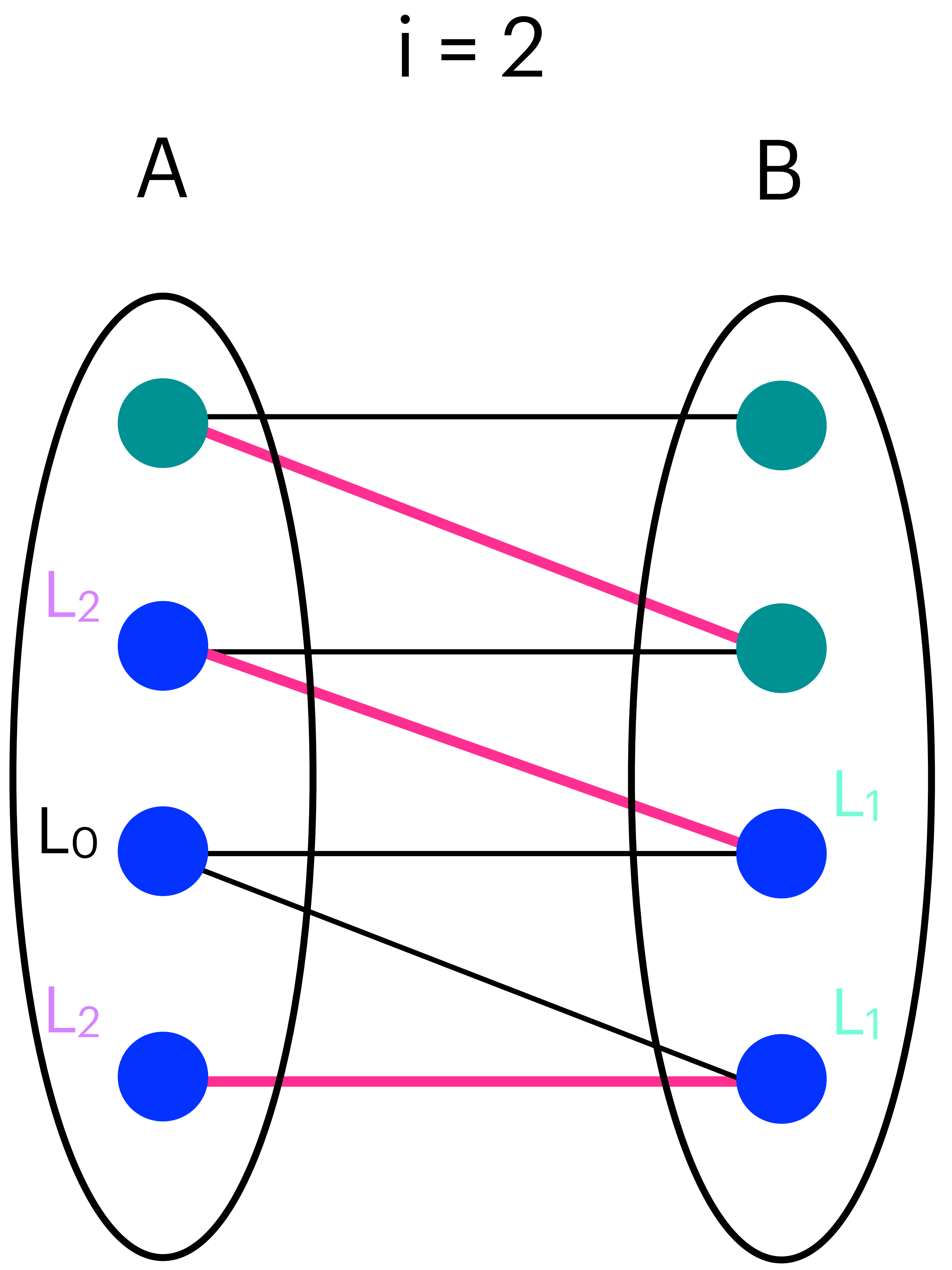
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

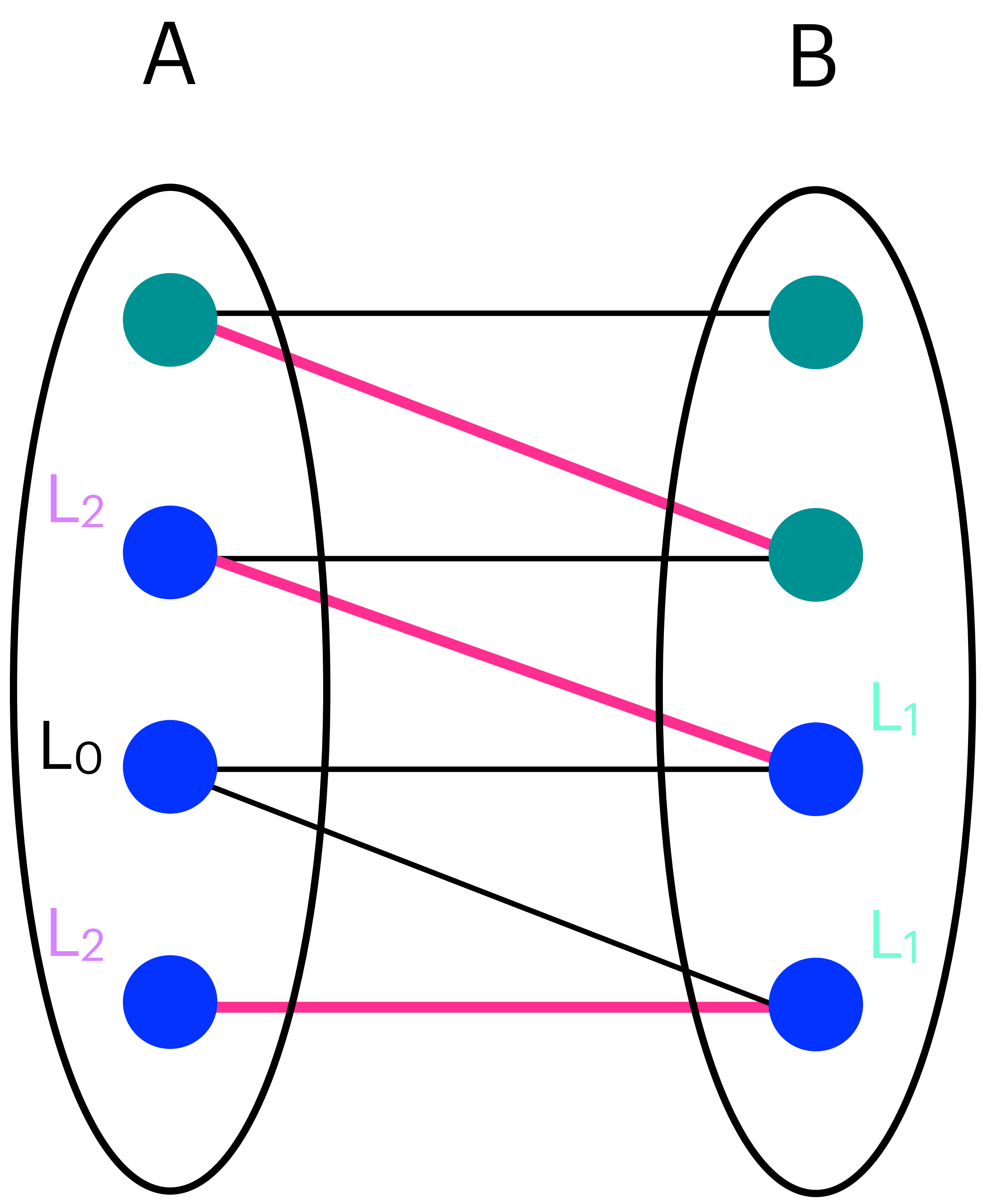
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 3



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

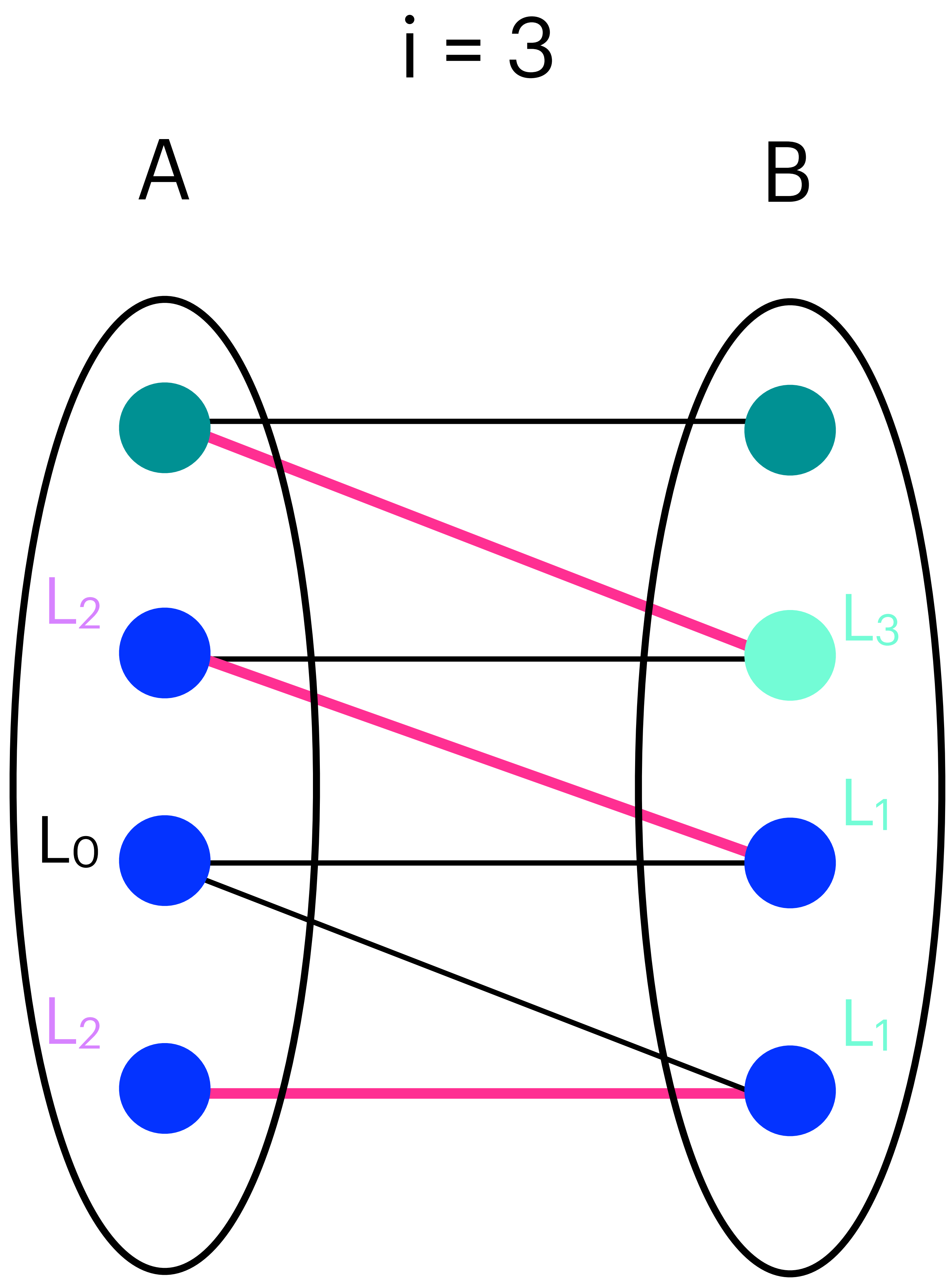
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

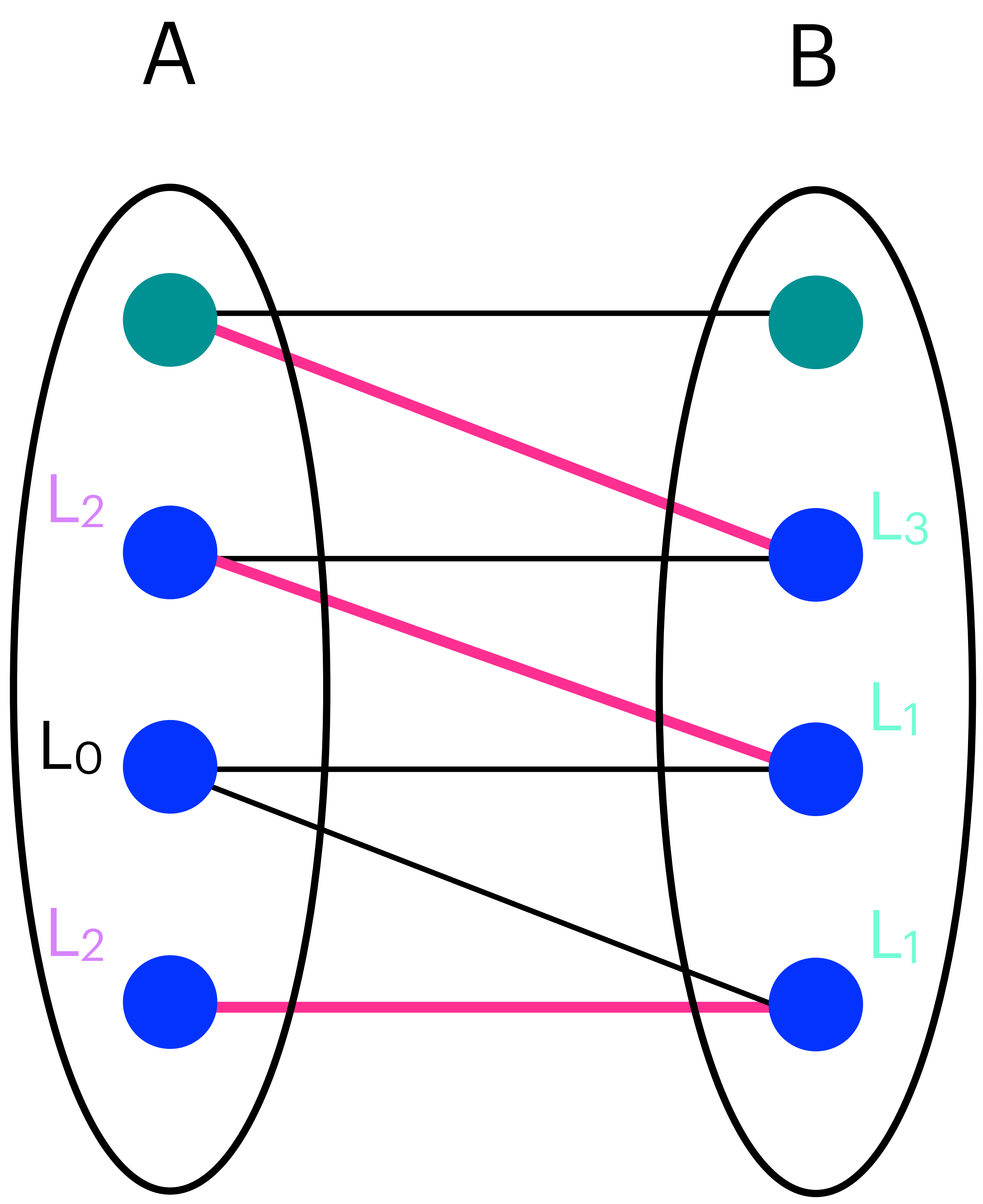
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 3



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

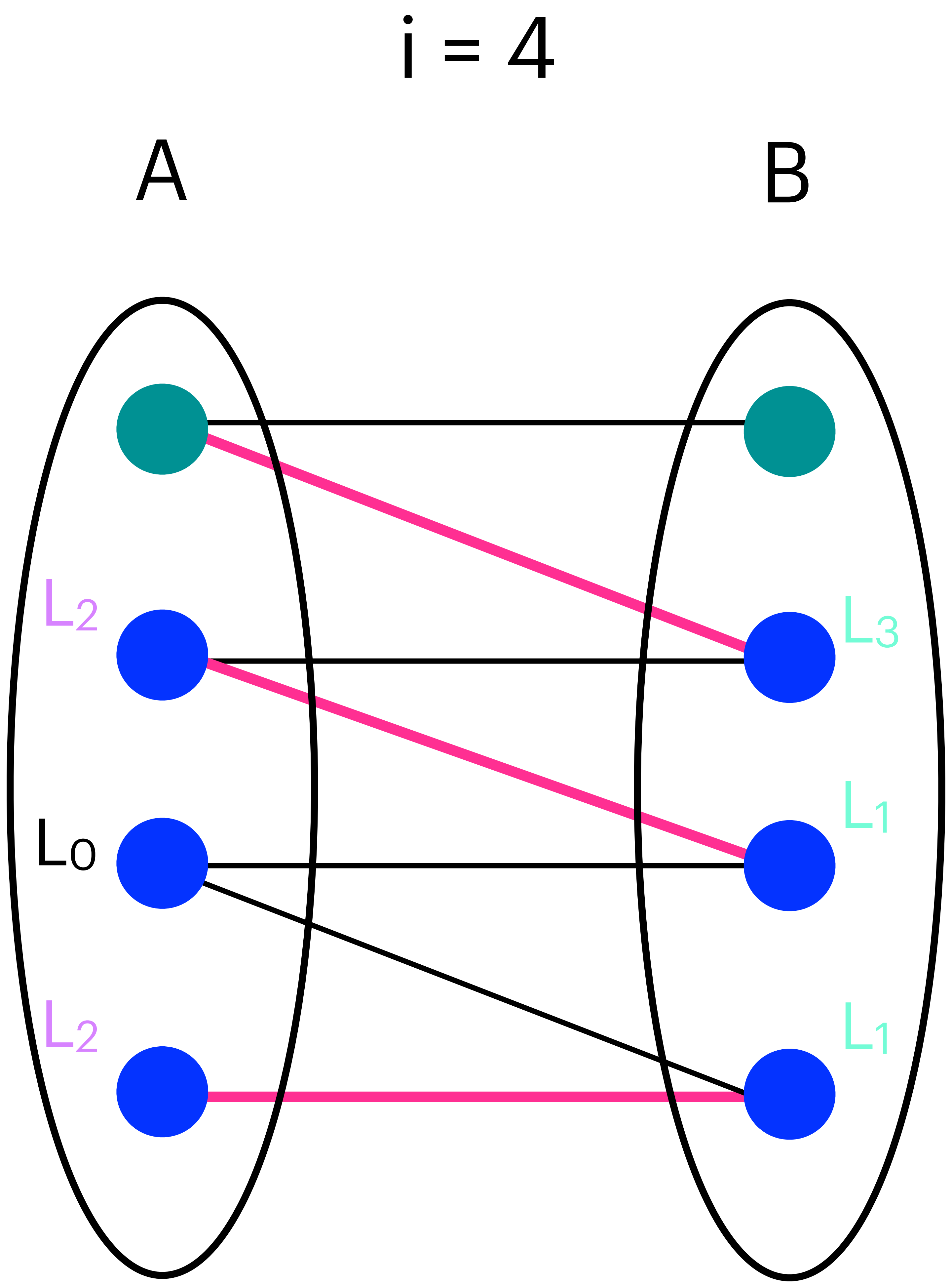
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

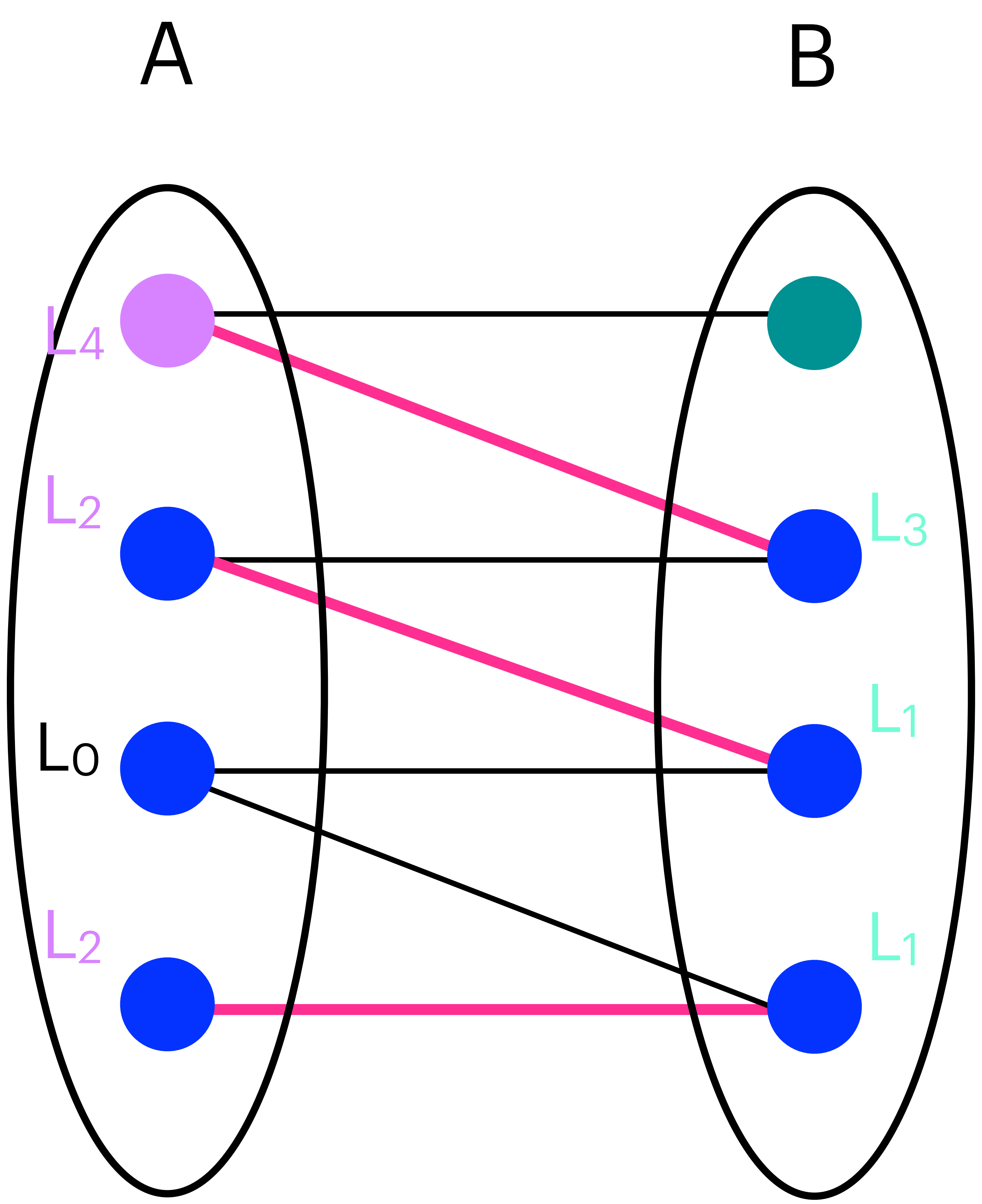
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 4



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

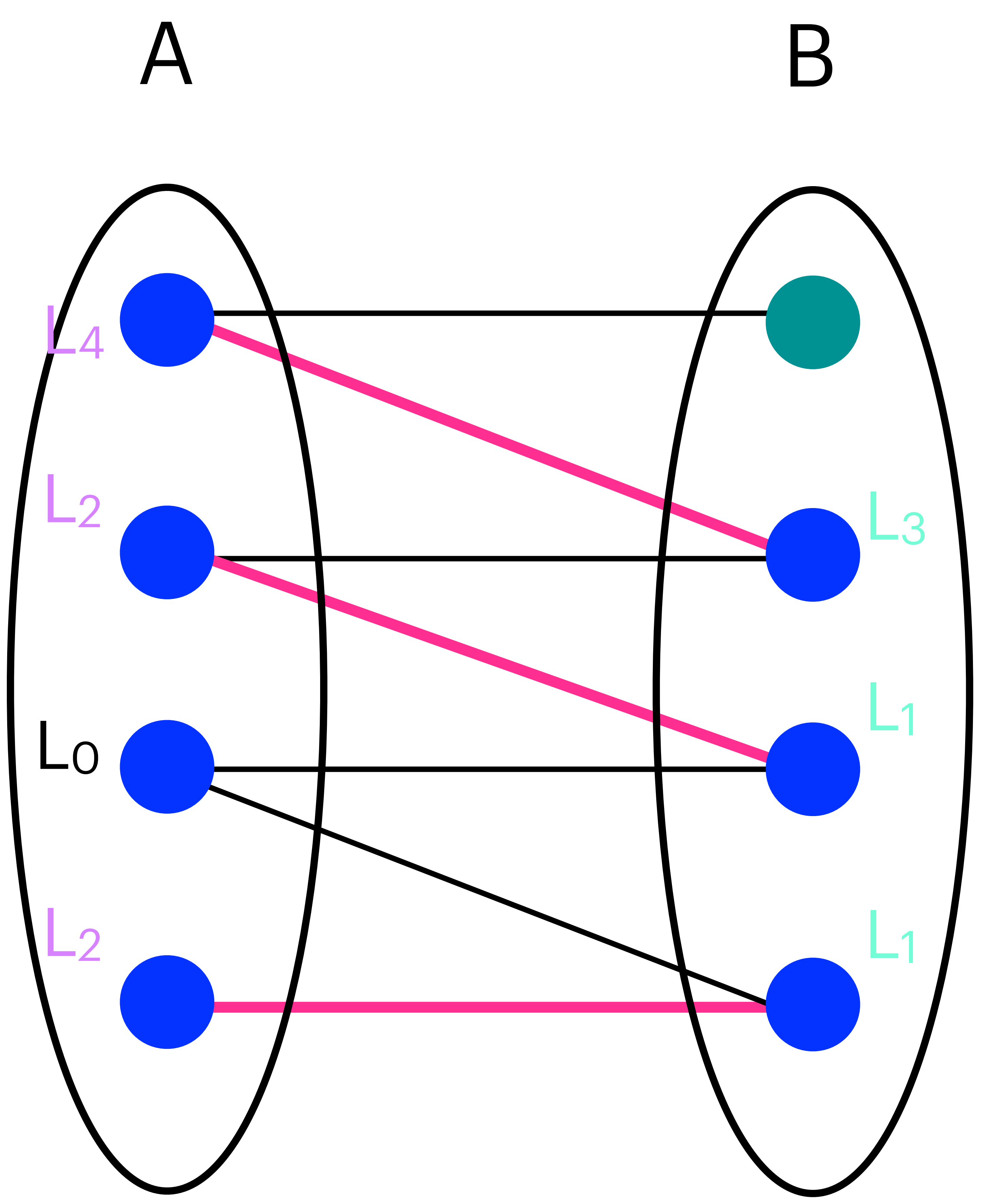
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 4



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

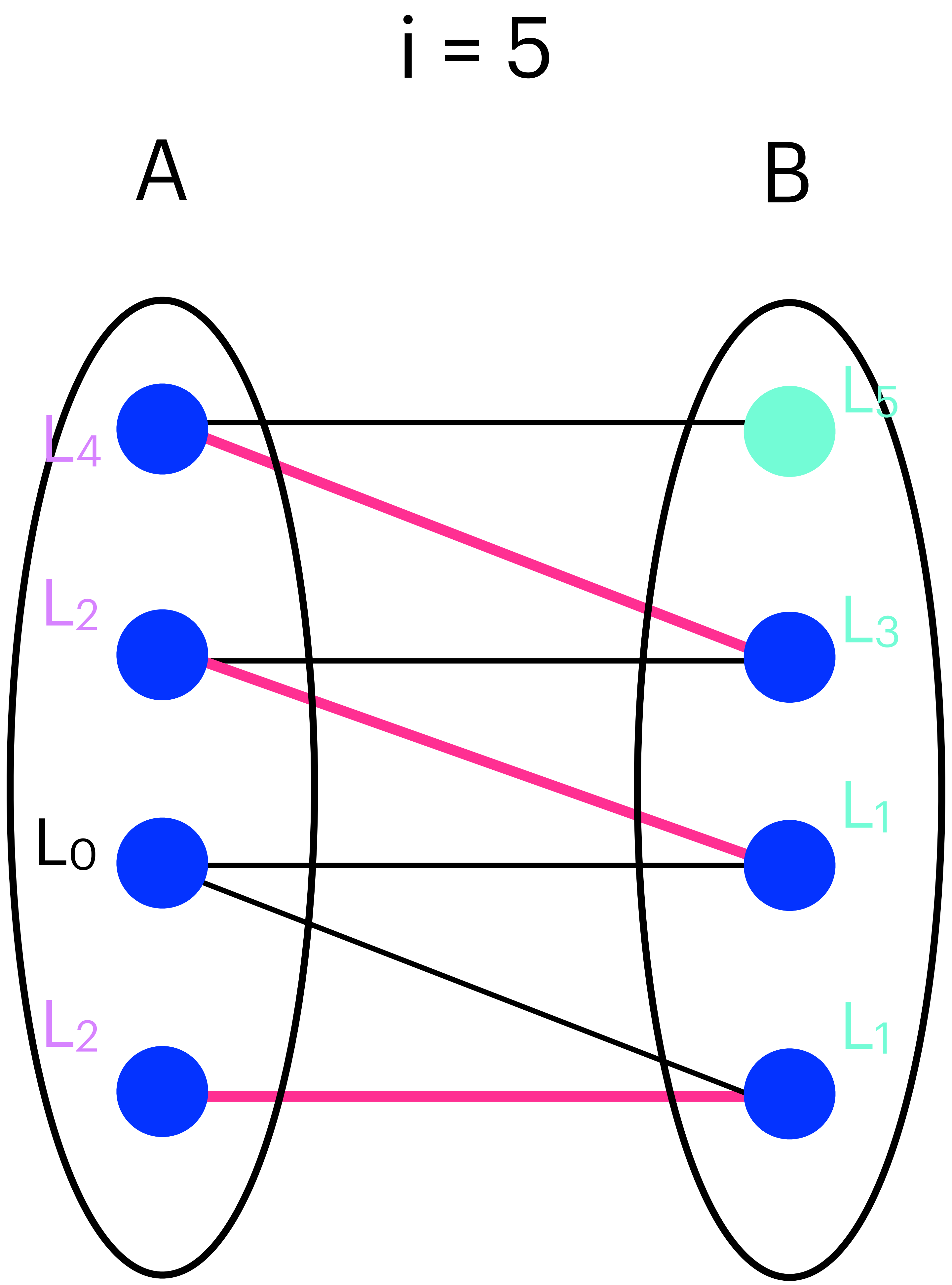
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

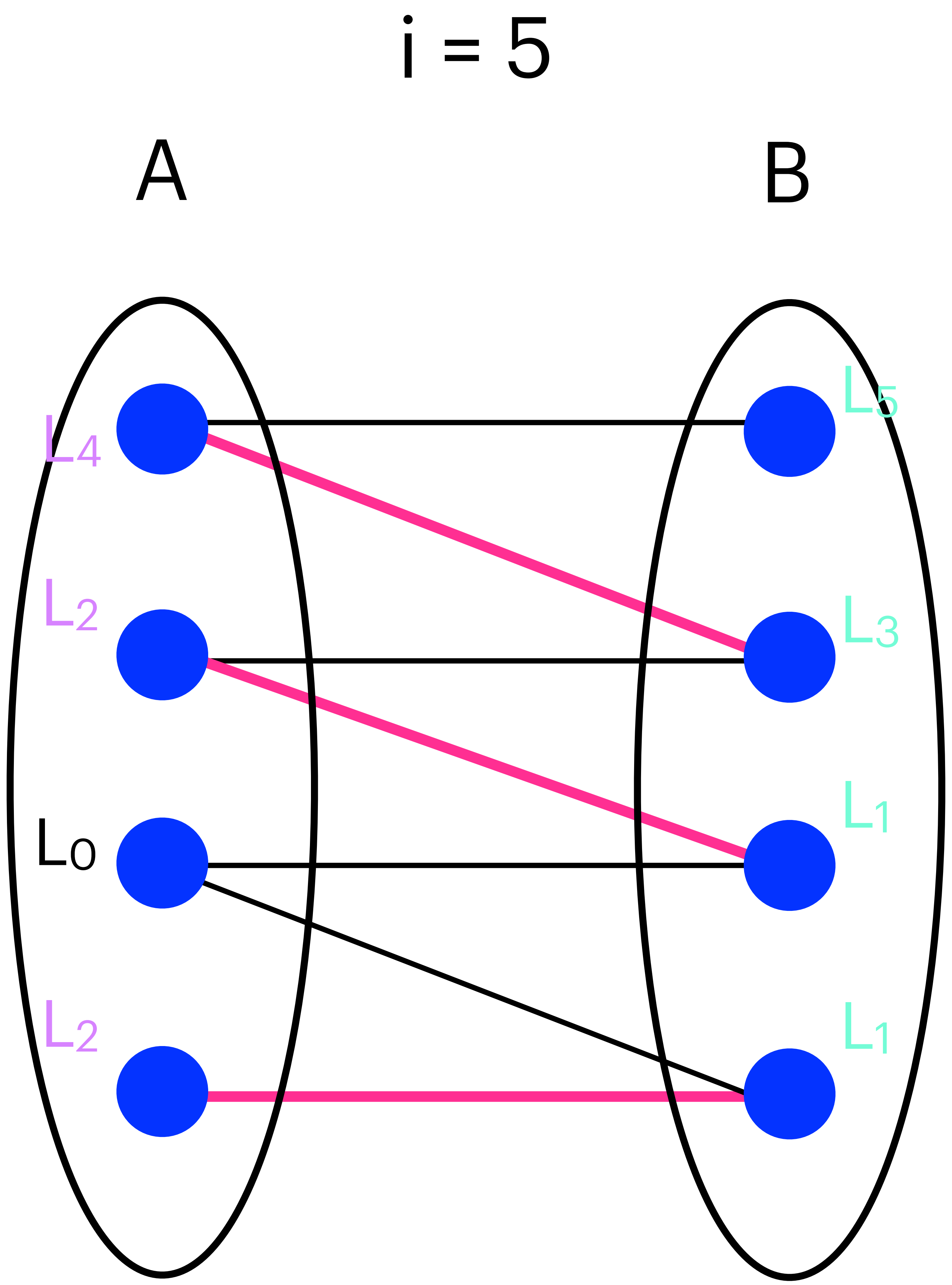
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

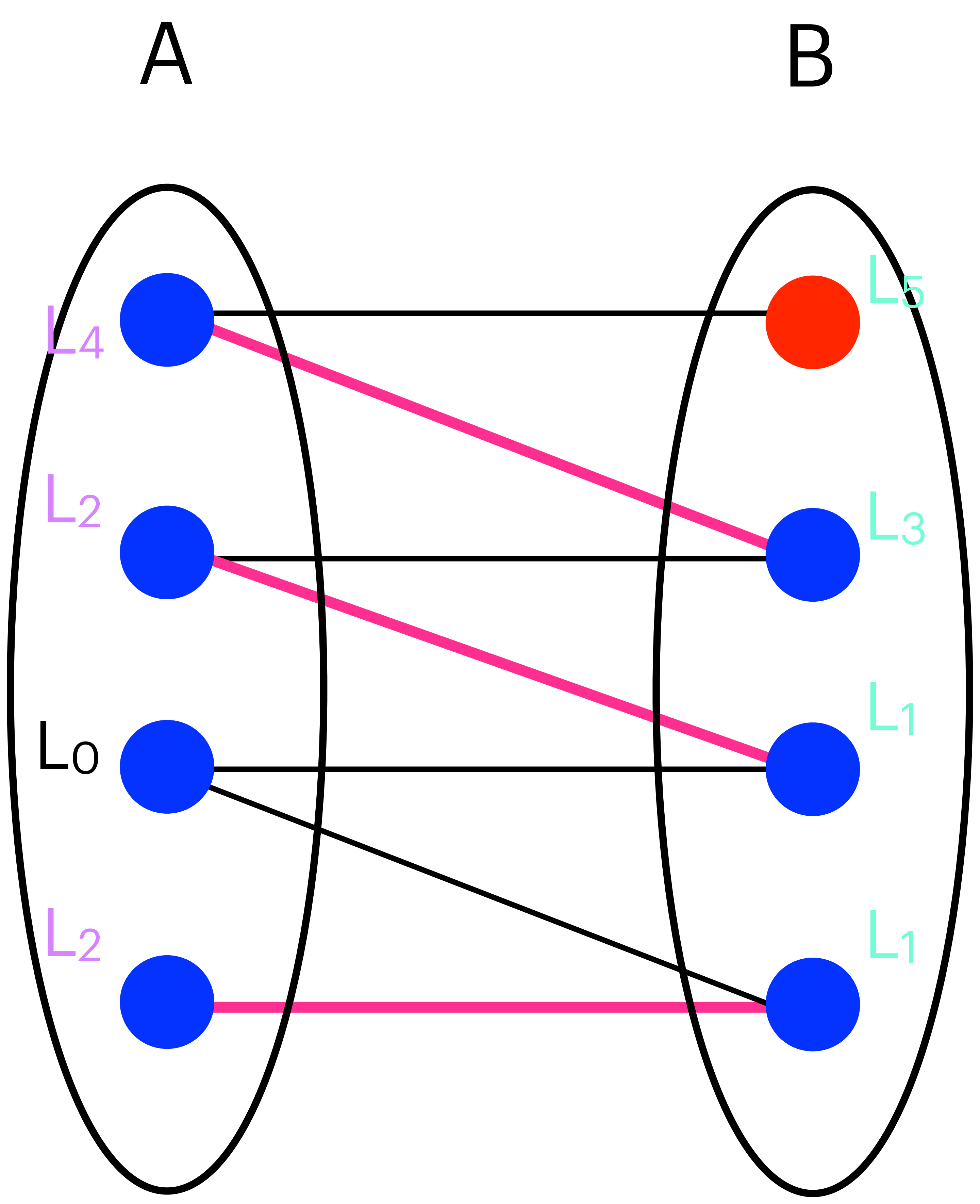
if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)

i = 5



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

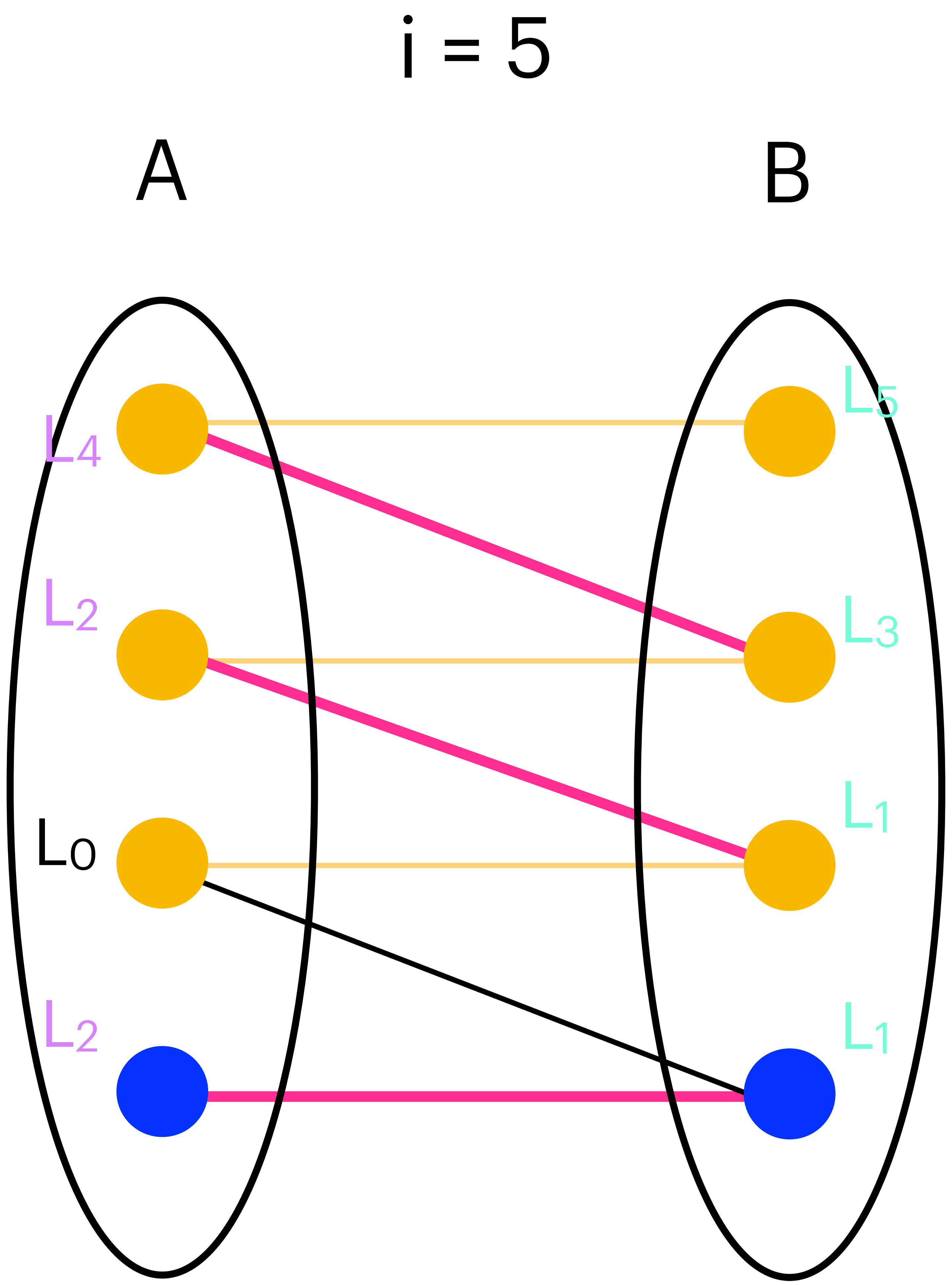
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Matching

BFS for augmenting paths

Input : A bipartite $G = (A \cup B, E)$, Matching M

Output : (shortest) augmenting path (if there is one)

Algorithm :

L_0 := {uncovered vertices from A}

Mark L_0 as visited

for i = 1 to n

if i is odd then

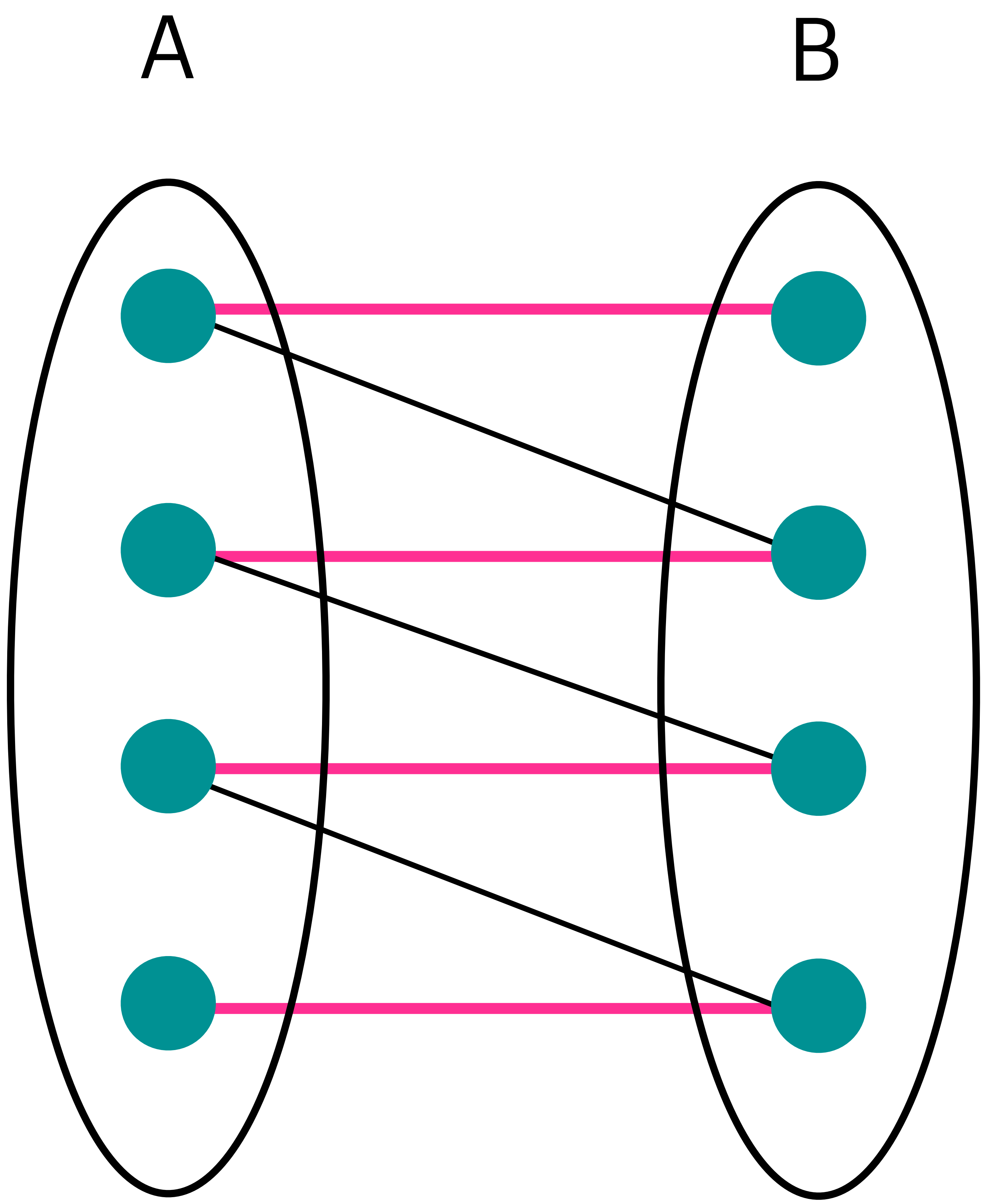
L_i := {unvisited neigbours of L_{i-1} via edges in E\M}

if i is even then

L_i := {unvisited neighbours of L_{i-1} via edges in M}

mark vertices from L_i as visited

if a vertex v in L_i is not covered : return path to v (backtracking) (M update)



Input : A bipartite $G = (A \cup B, E)$

Output : Maximum Matching M

Matching

Improvement : Hopcroft Karp Algorithm

Algorithm :

Start with $M = \emptyset$

while $\exists$ augmenting path P with BFS

$M = M \oplus P$

return M

Hopcroft-Karp :

Start with $M = \emptyset$

while $\nexists$ augmenting path P

k := length of the shortest augmenting path

find more disjoint augmenting paths of length k

until we have a inclusion-maximal set S of those paths

for all P in S :

$M = M \oplus P$

$O(|V|^{1/2} \cdot (|V|+|E|))$

Let's take a break

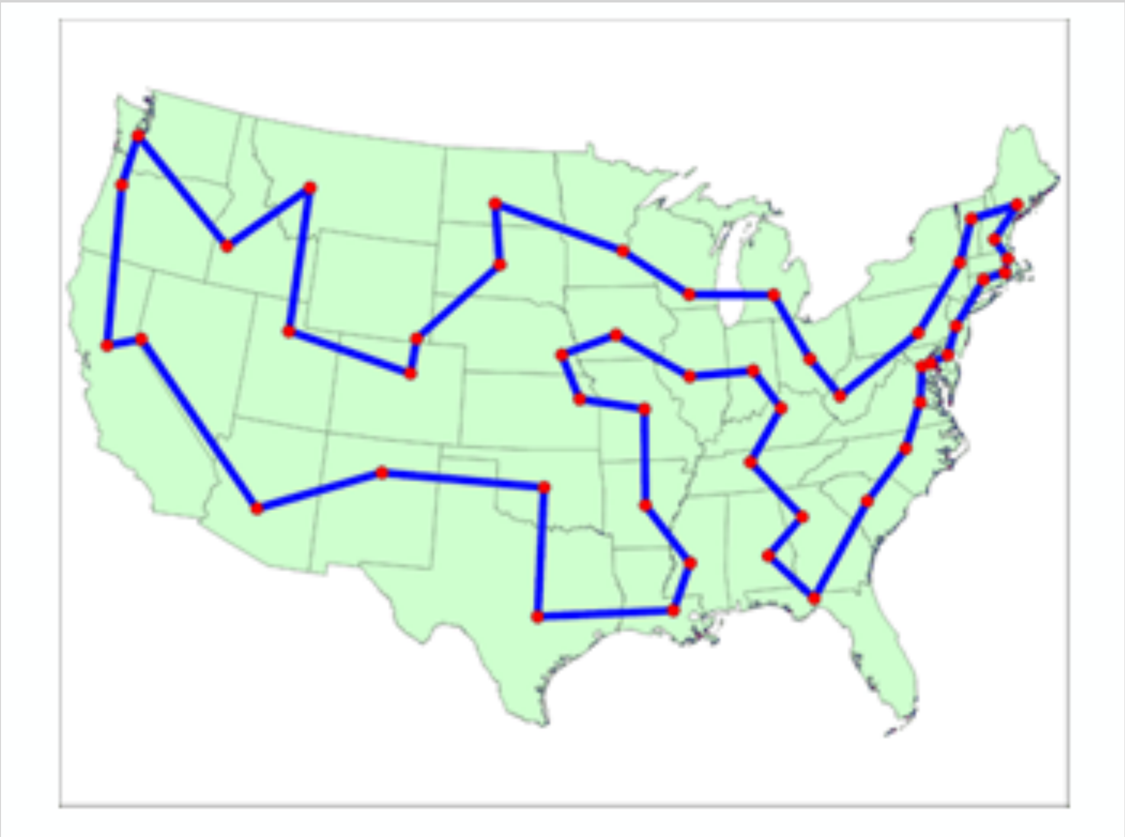
A&W 😊🧑🏻‍💻

WhatsApp group

TSP II

Metric TSP : 2-Approximation

Problem Description



Given :

- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$
- l satisfies the triangle inequality $l(x, z) \leq l(x, y) + l(y, z)$

To find :

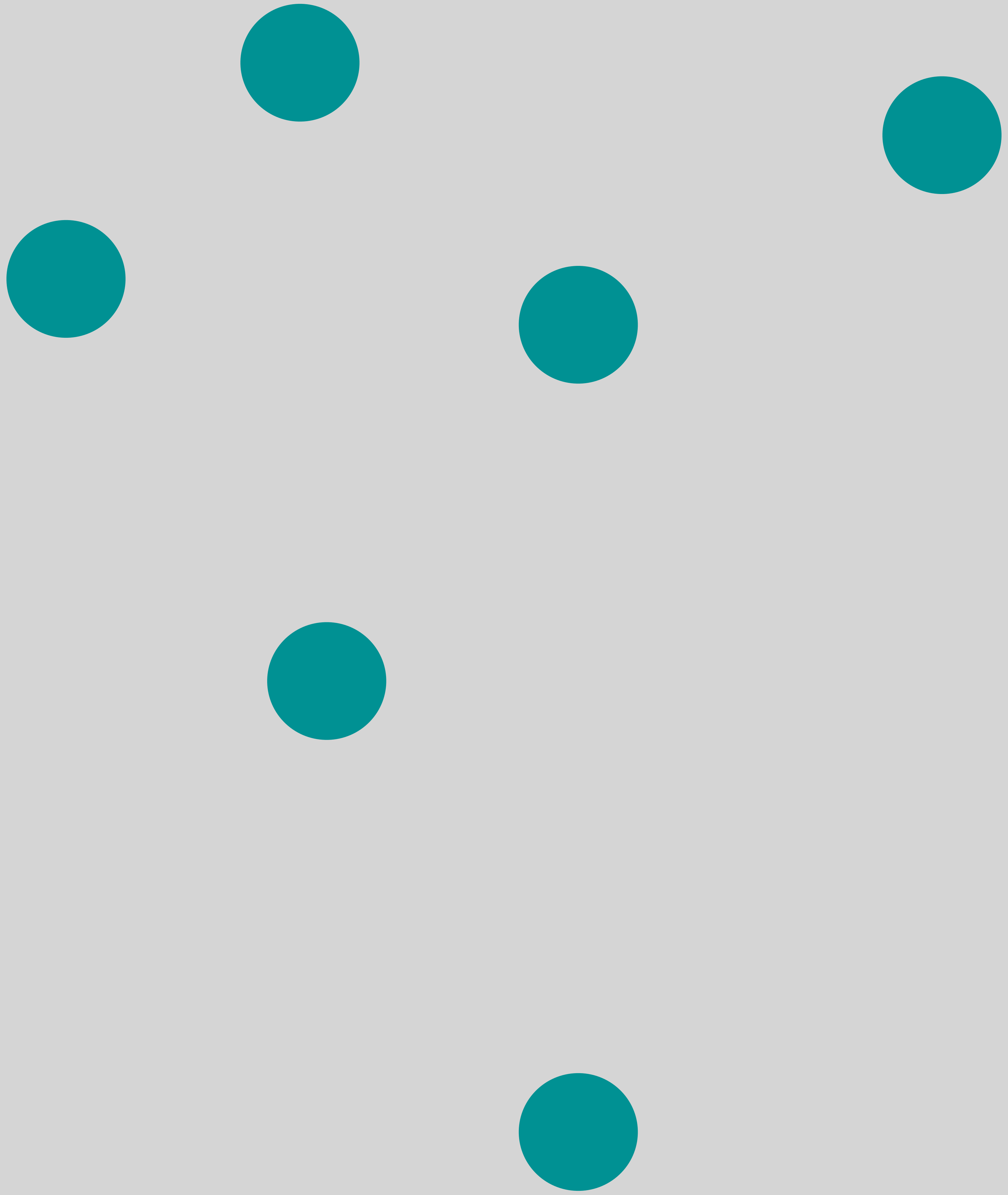
- Hamiltonian Cycle C s.t.

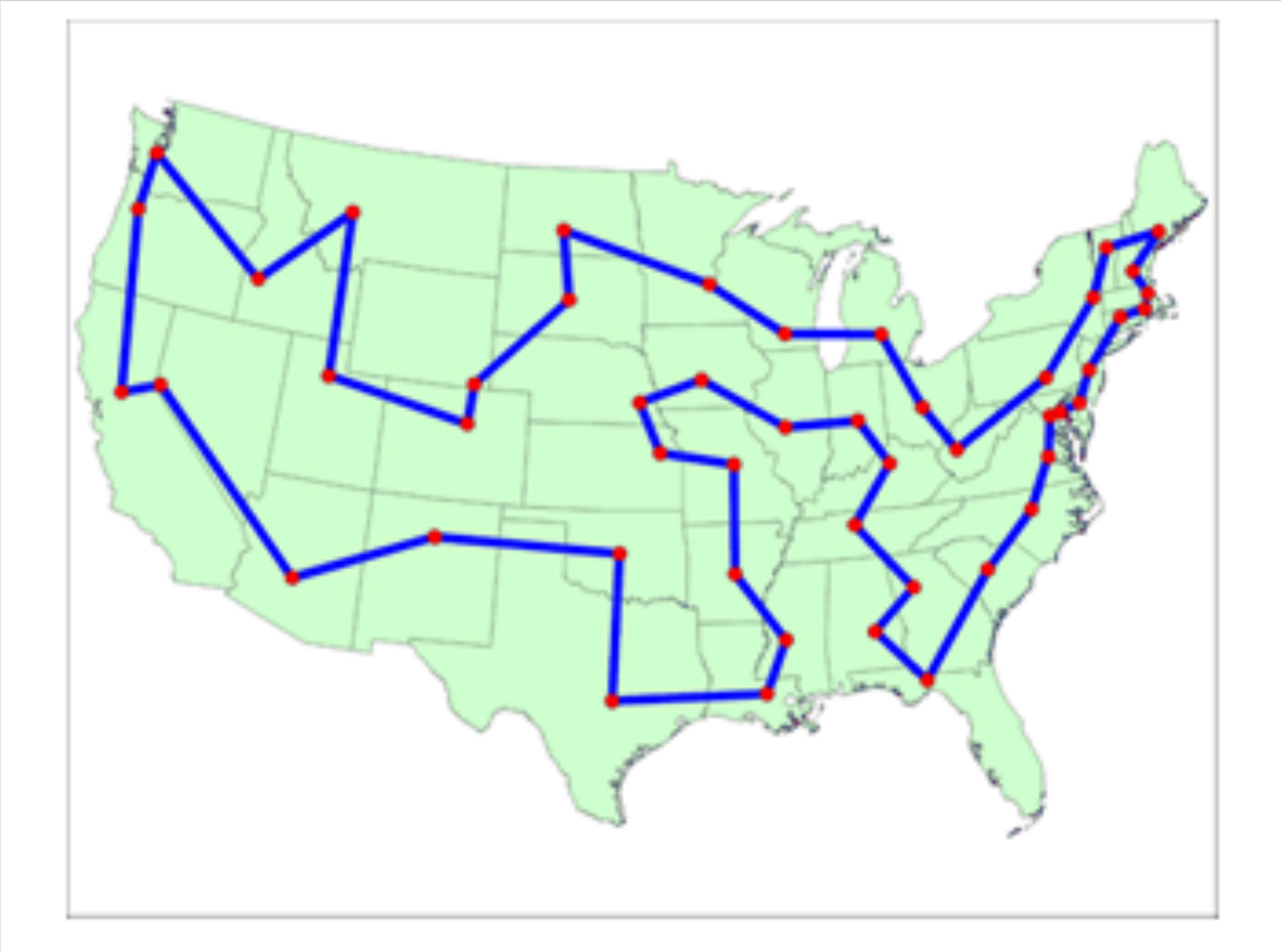
$$l(C) \leq 2\, l(OPT)$$

where $OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 2-Approximation

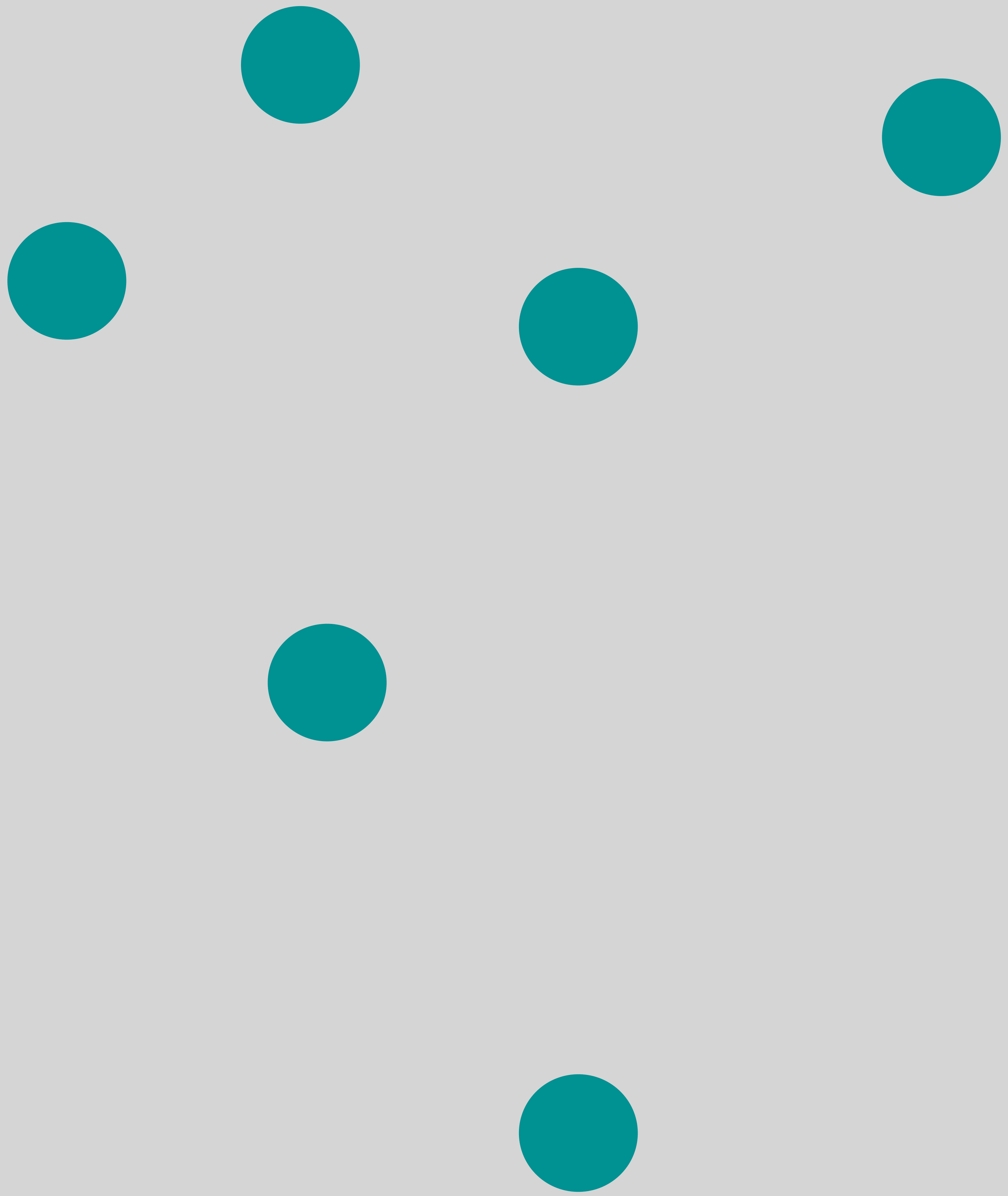
Algorithm





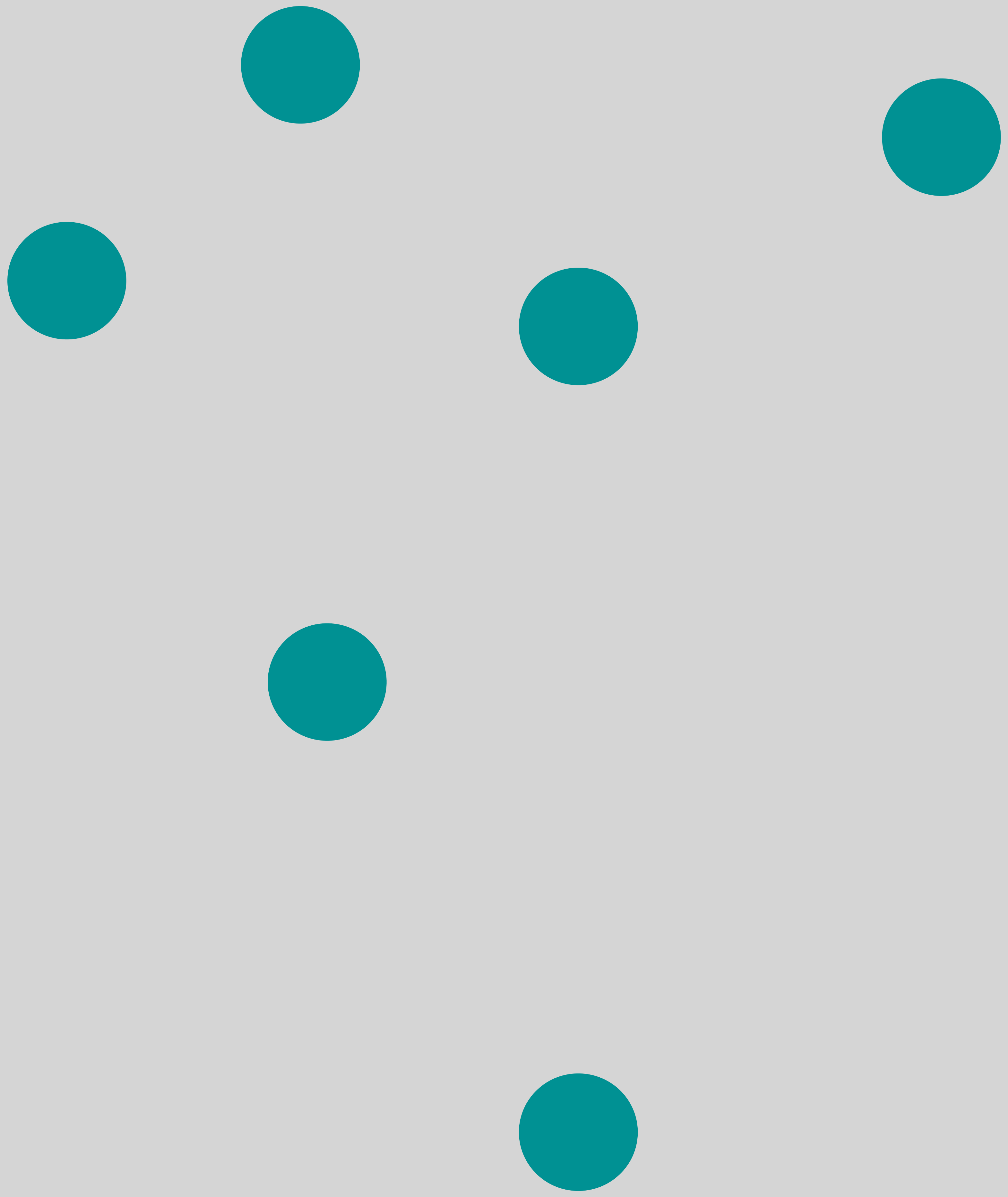
Metric TSP : 2-Approximation

Algorithm



Metric TSP : 2-Approximation

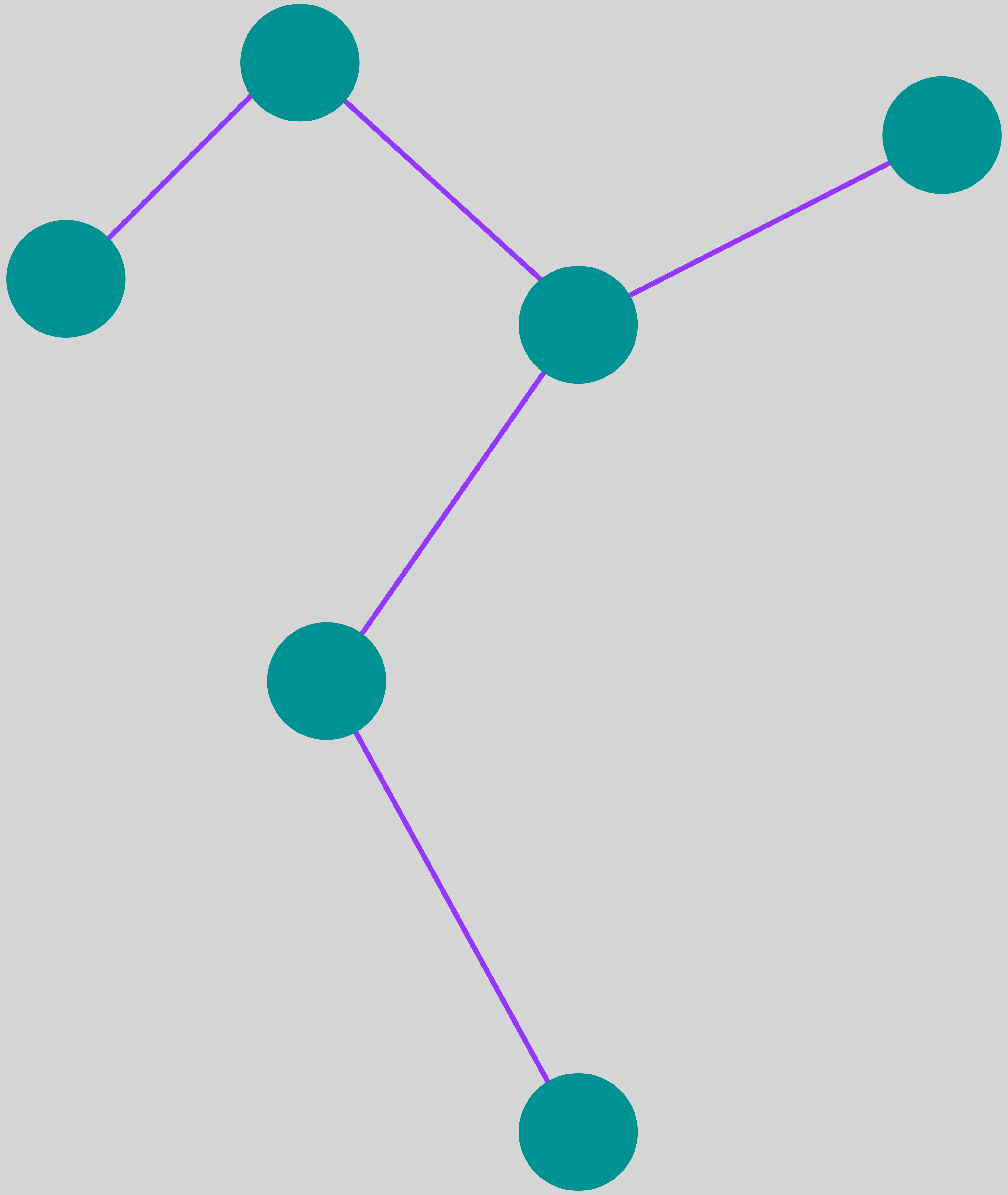
Algorithm



1. Find the MST T

Metric TSP : 2-Approximation

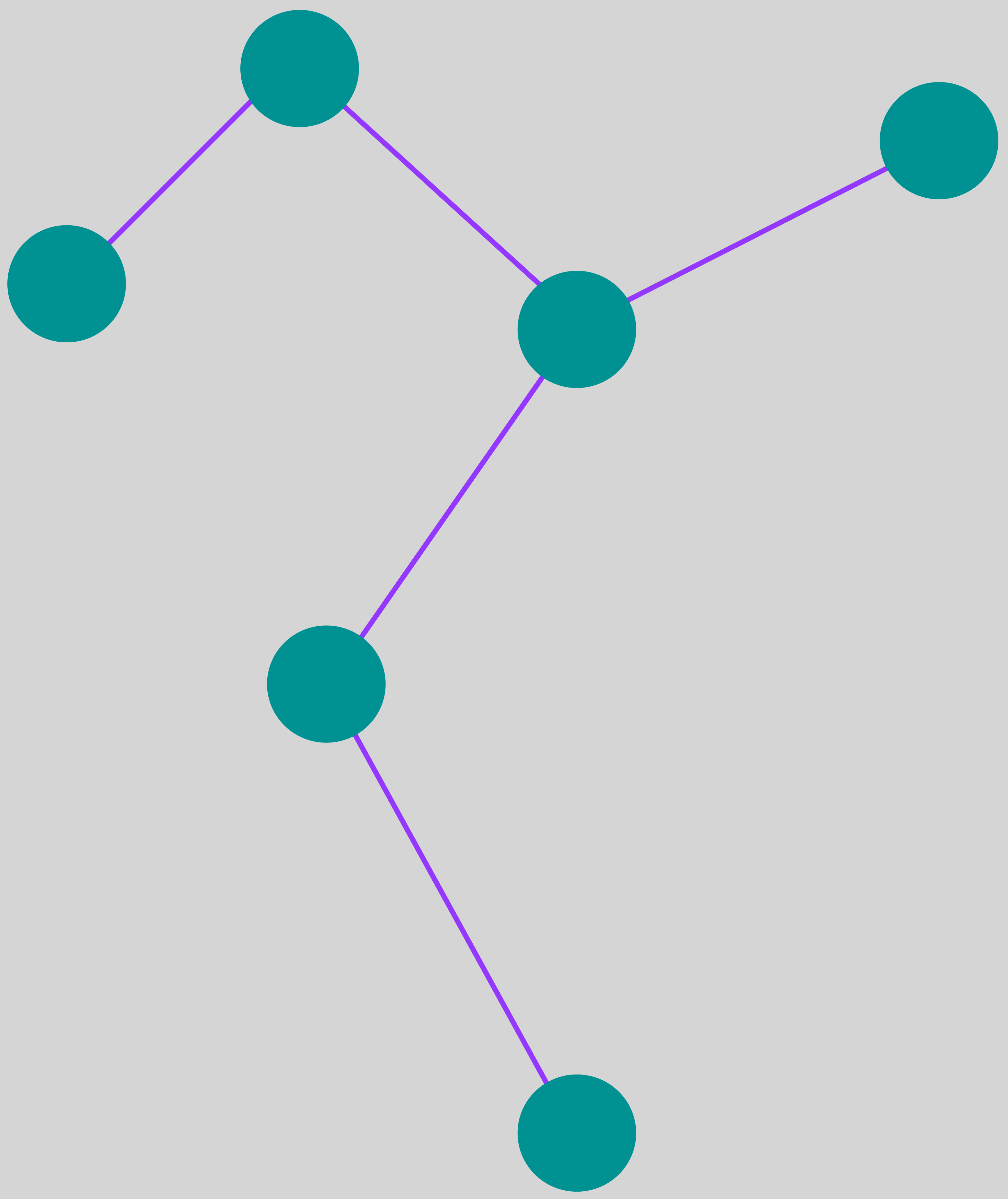
Algorithm



1. Find the MST T

Metric TSP : 2-Approximation

Algorithm

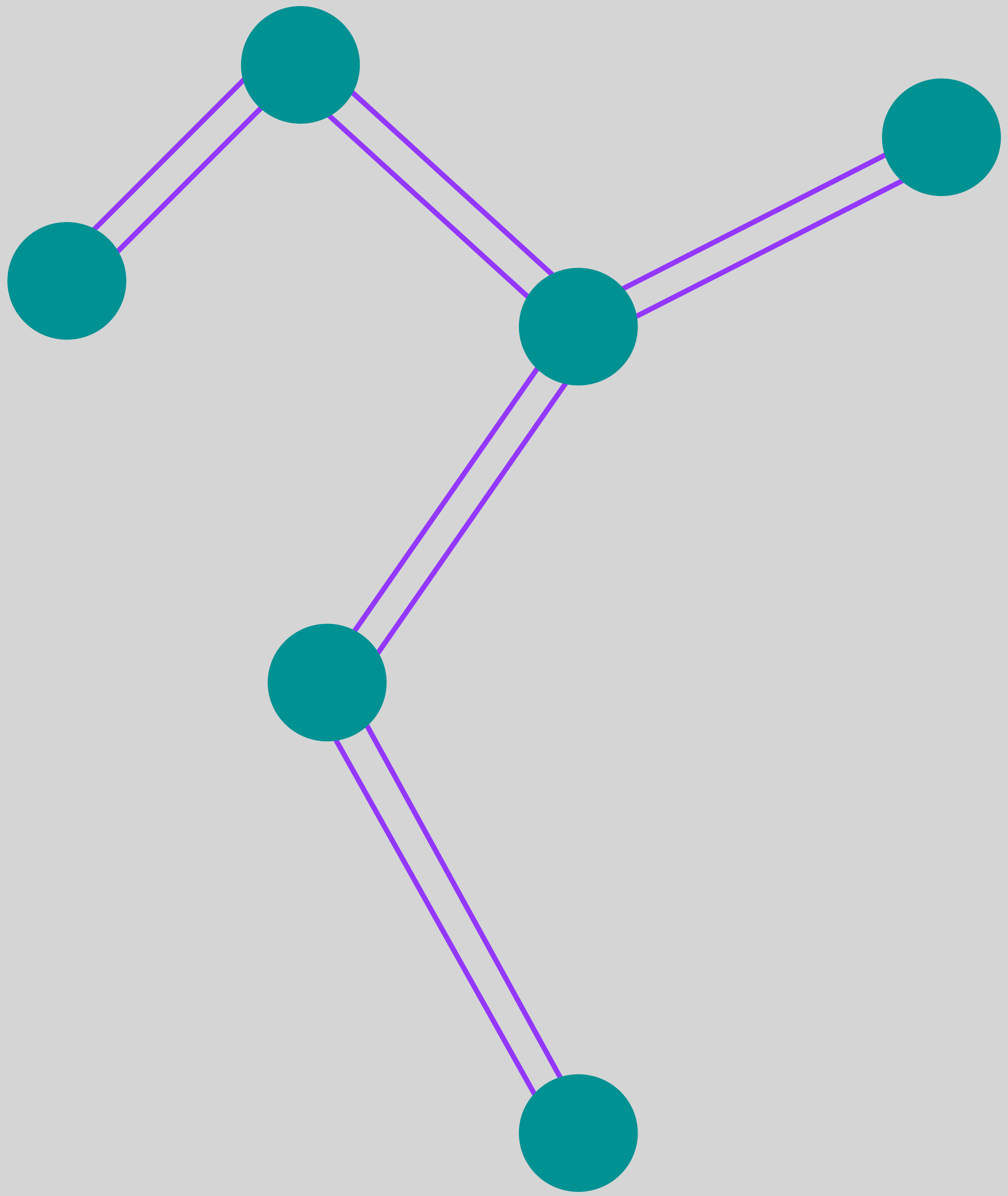


1. Find the MST T

2. Duplicate all edges of T

Metric TSP : 2-Approximation

Algorithm

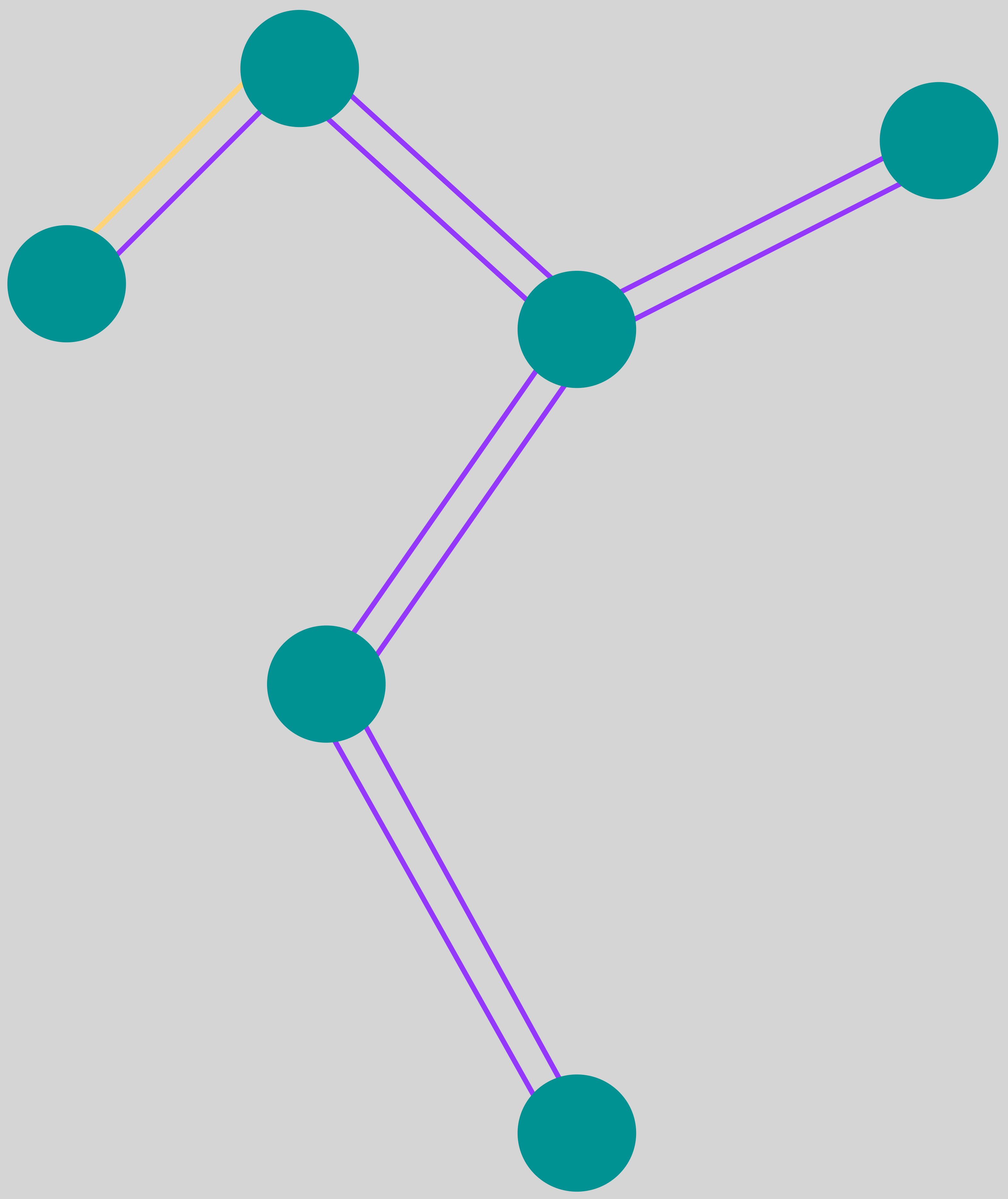


1. Find the MST T

2. Duplicate all edges of T

Metric TSP : 2-Approximation

Algorithm



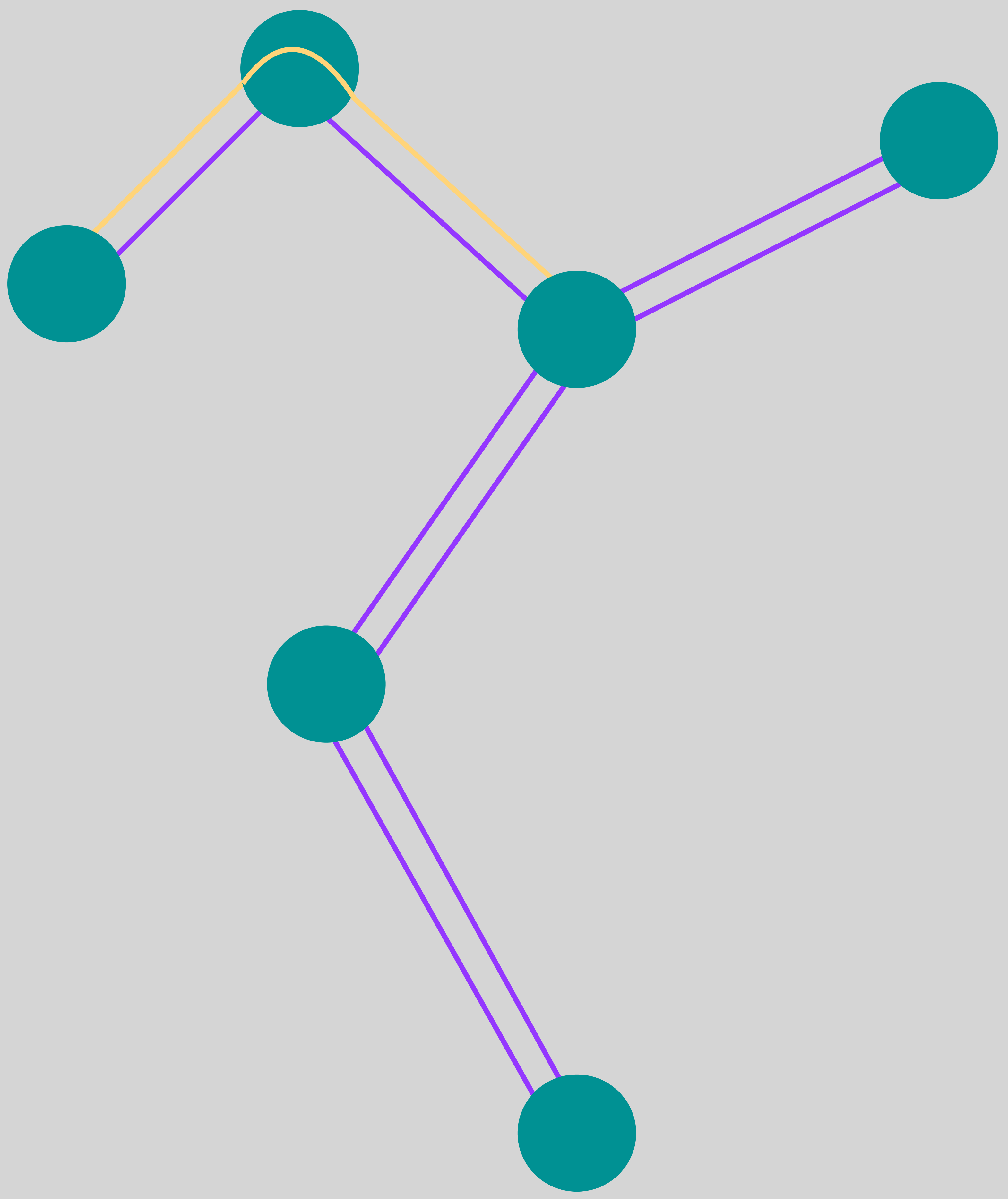
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



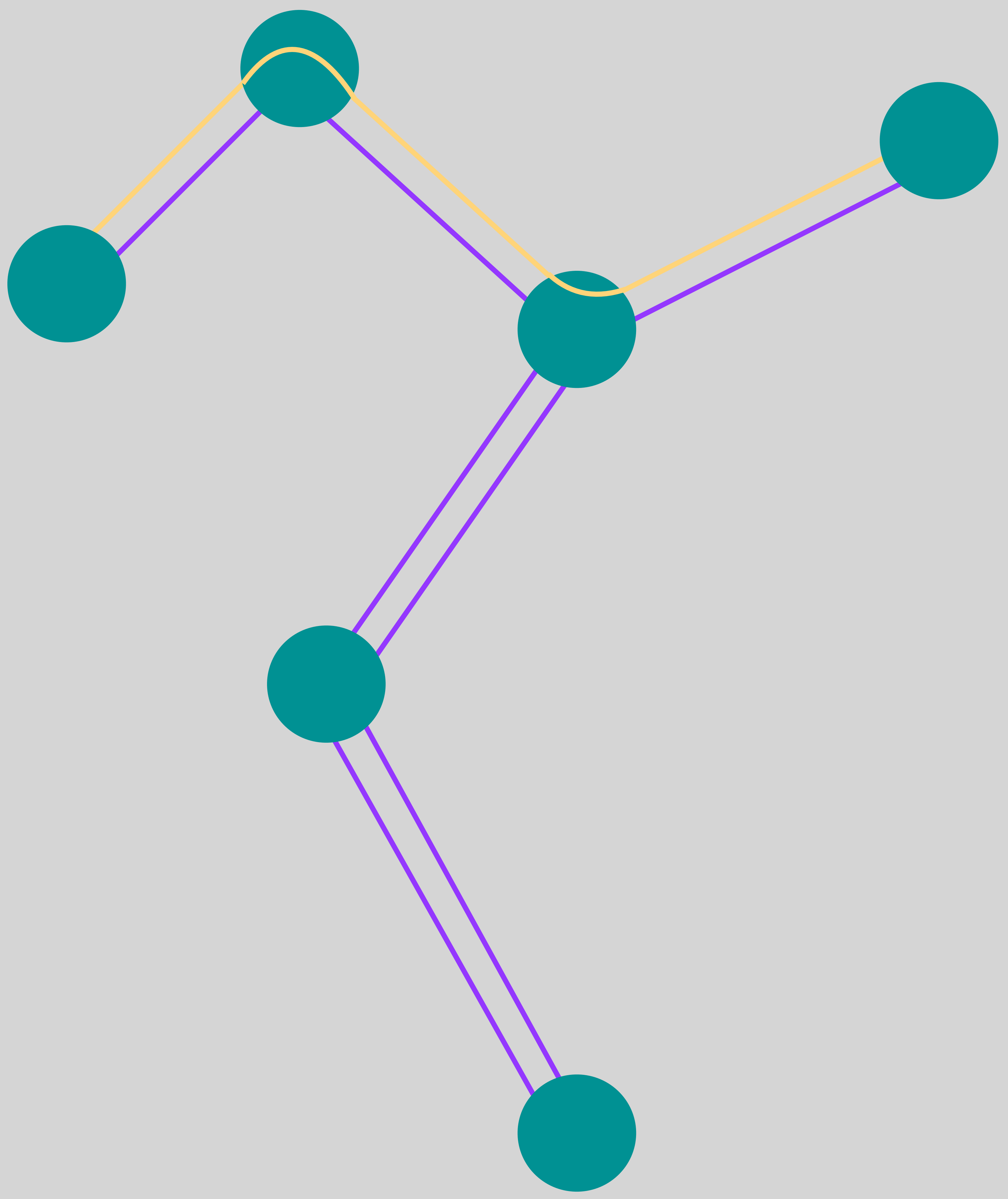
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



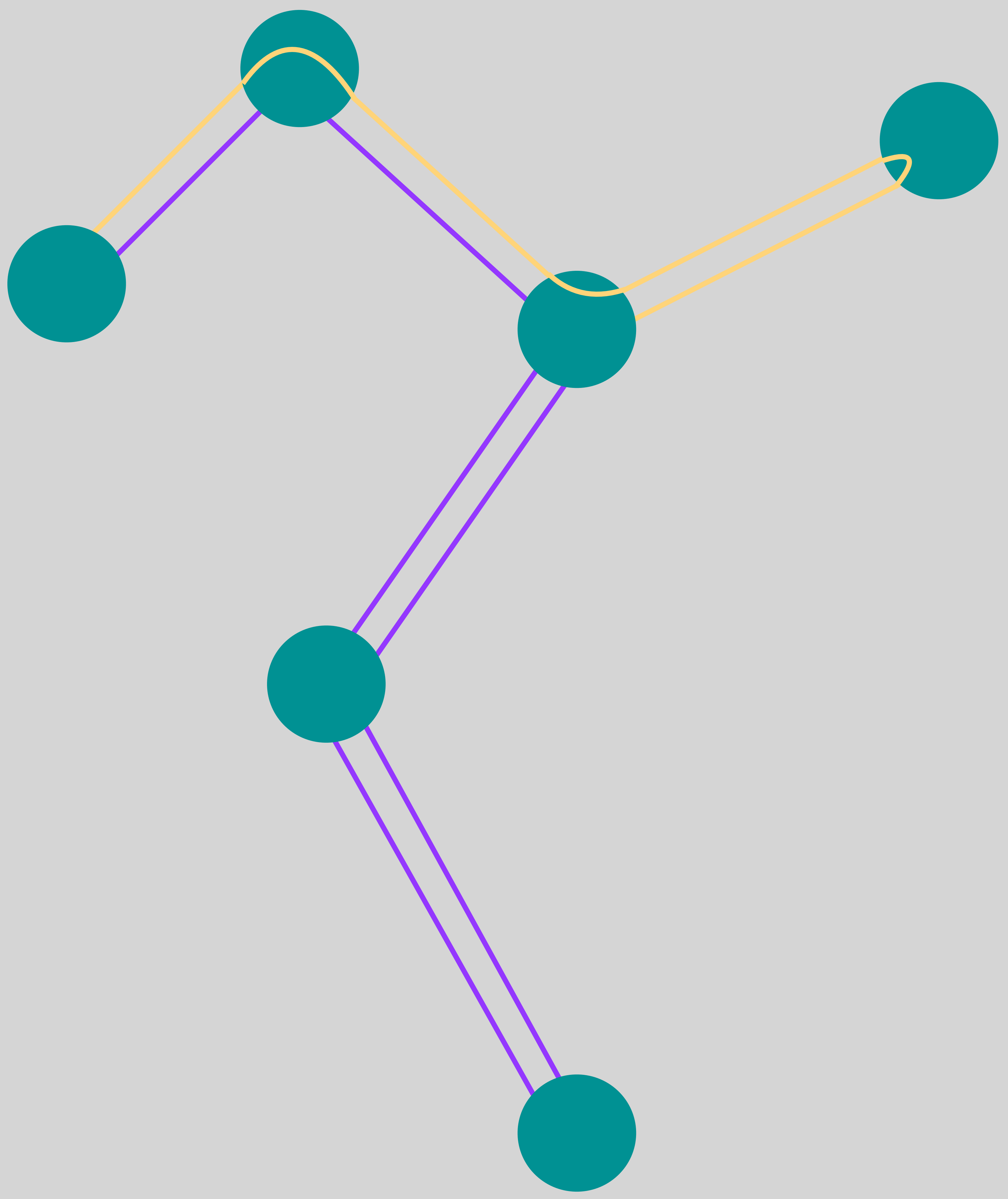
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



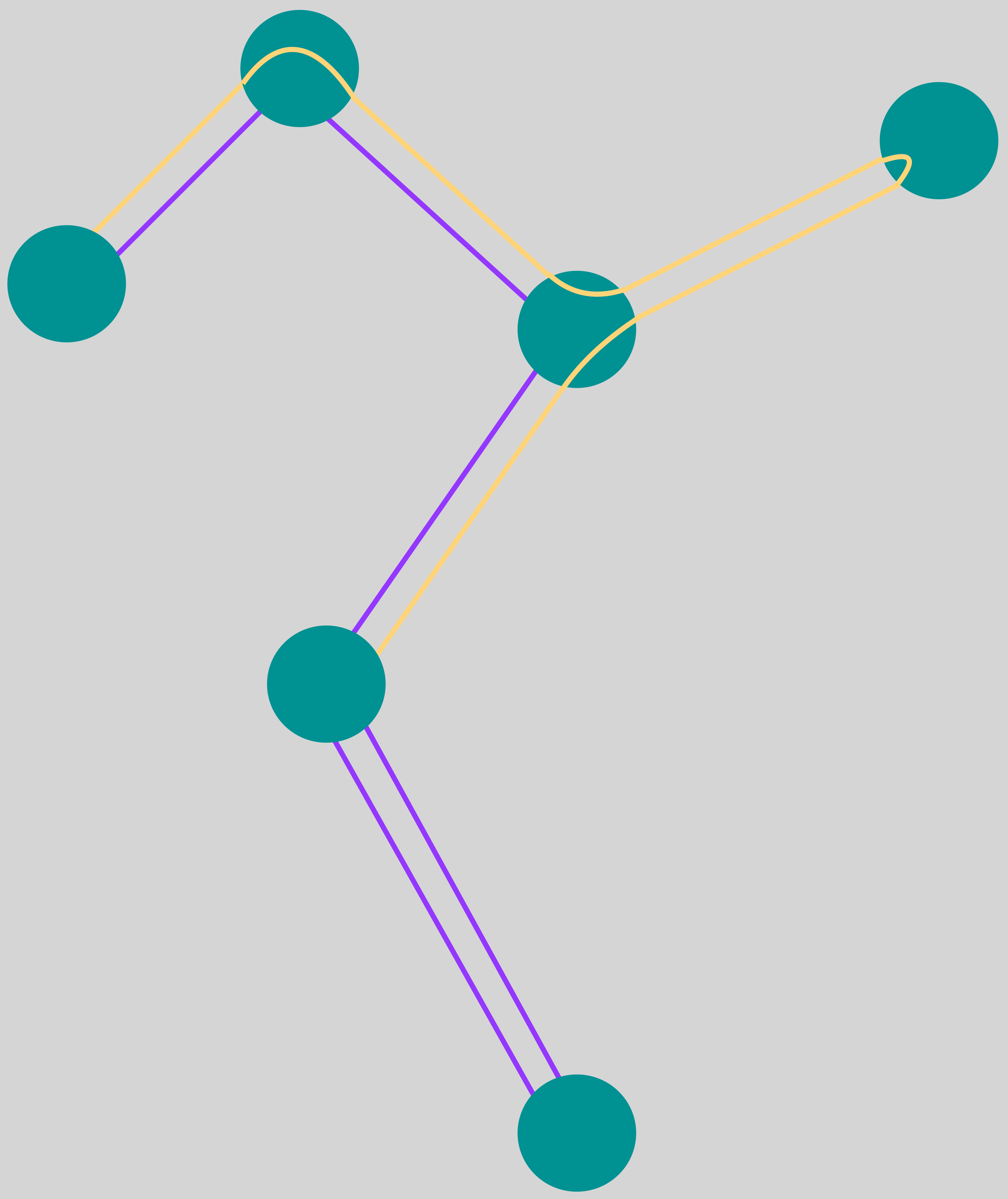
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



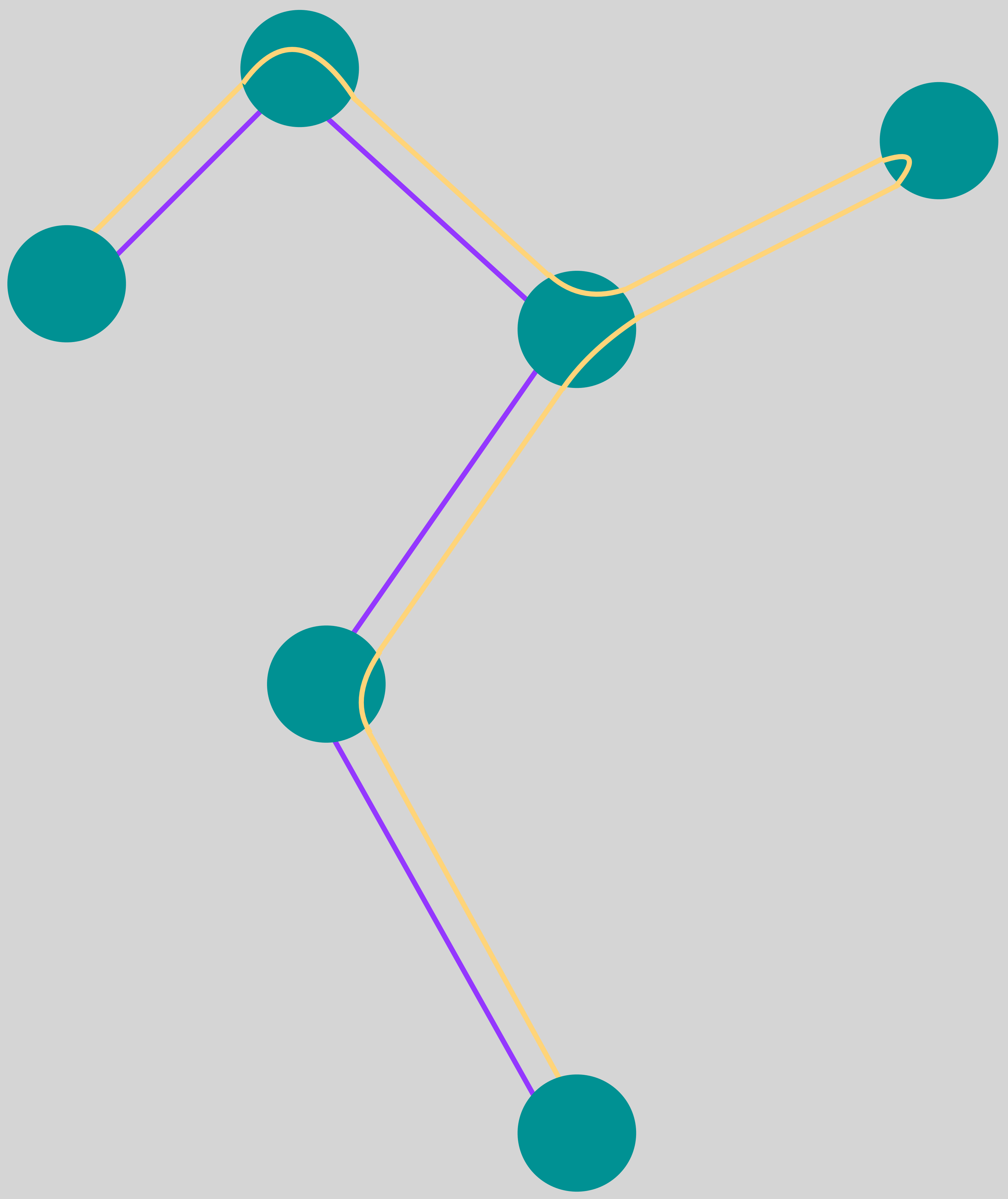
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



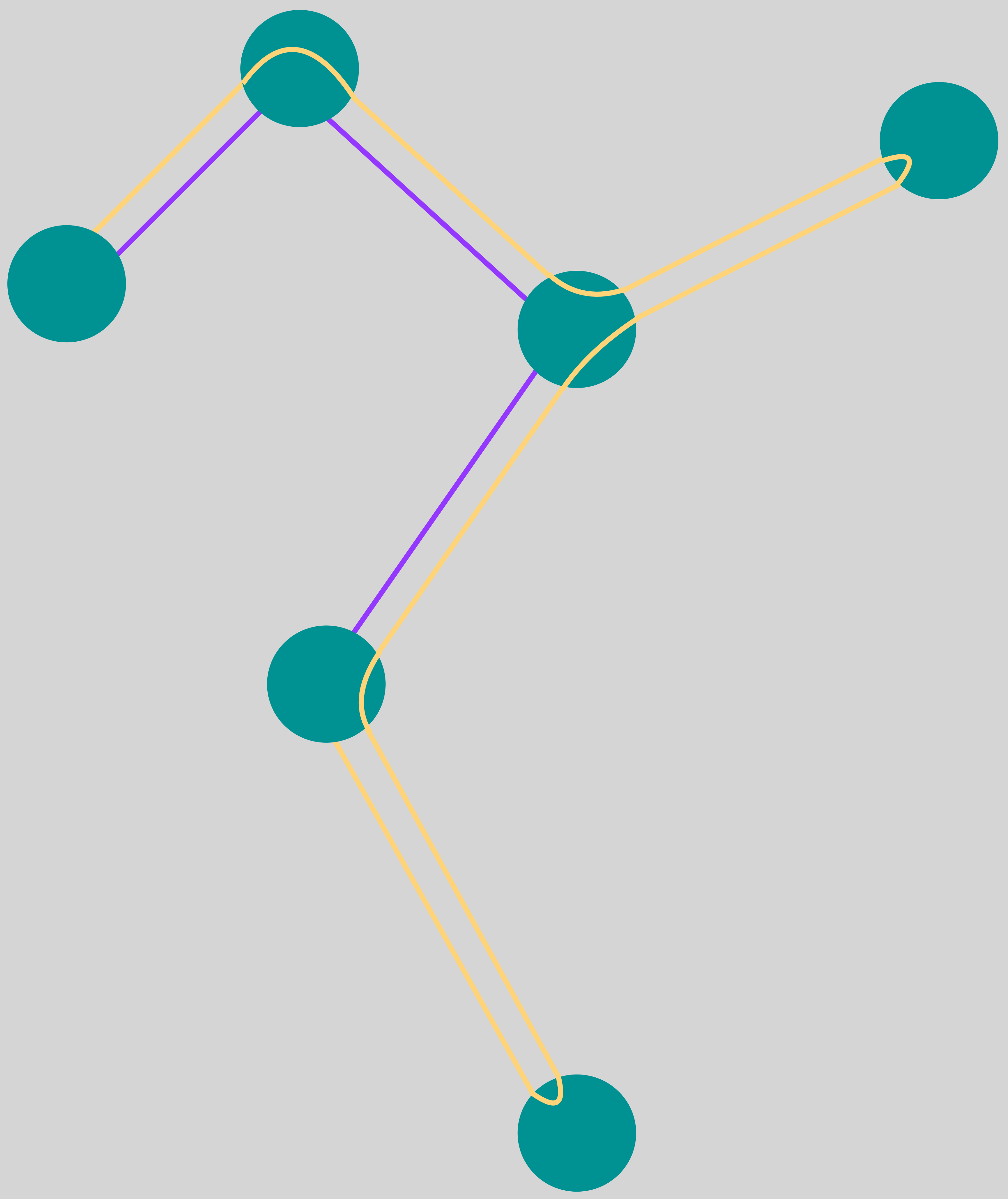
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



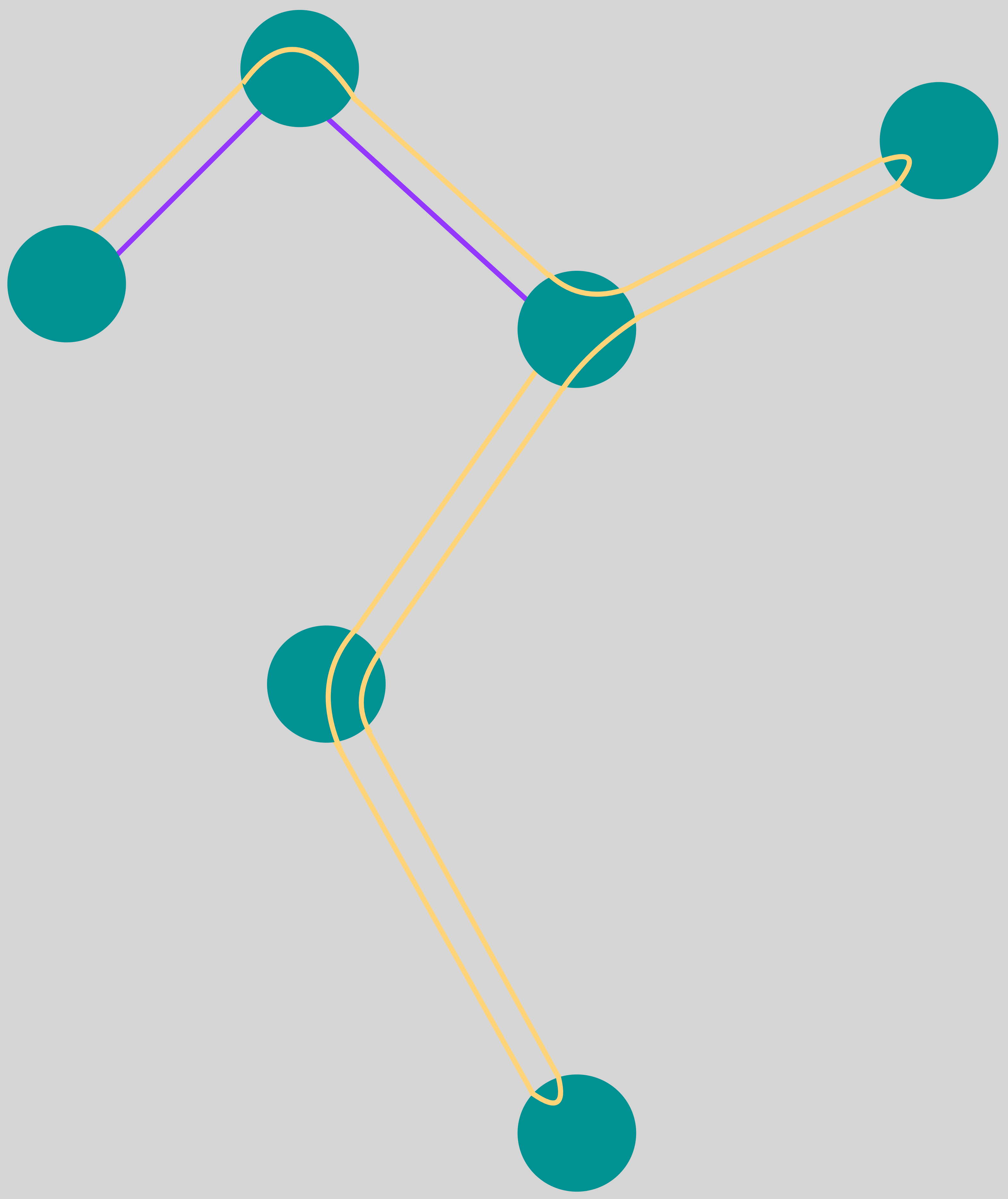
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

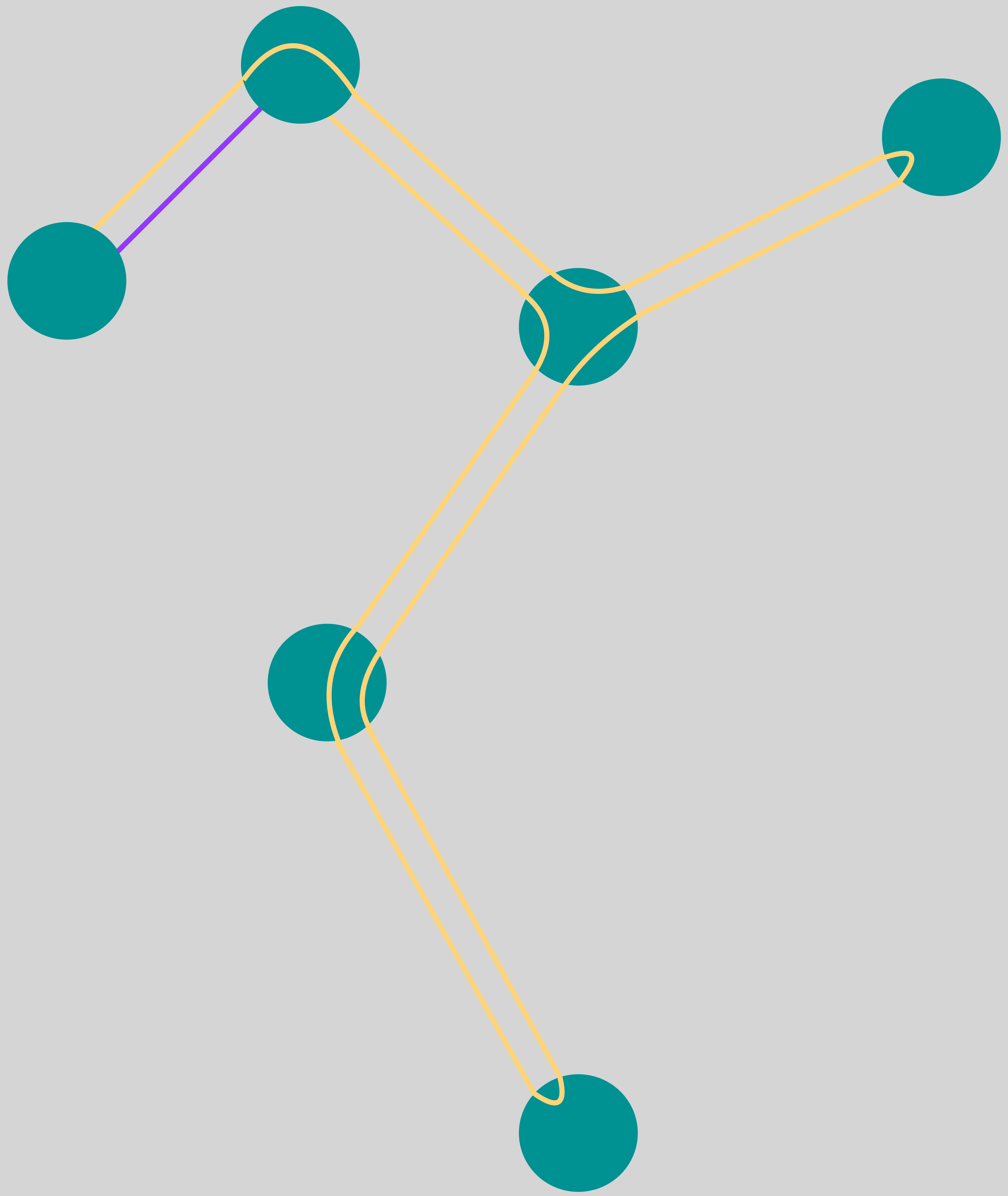
Algorithm



1. Find the MST T
2. Duplicate all edges of T
3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



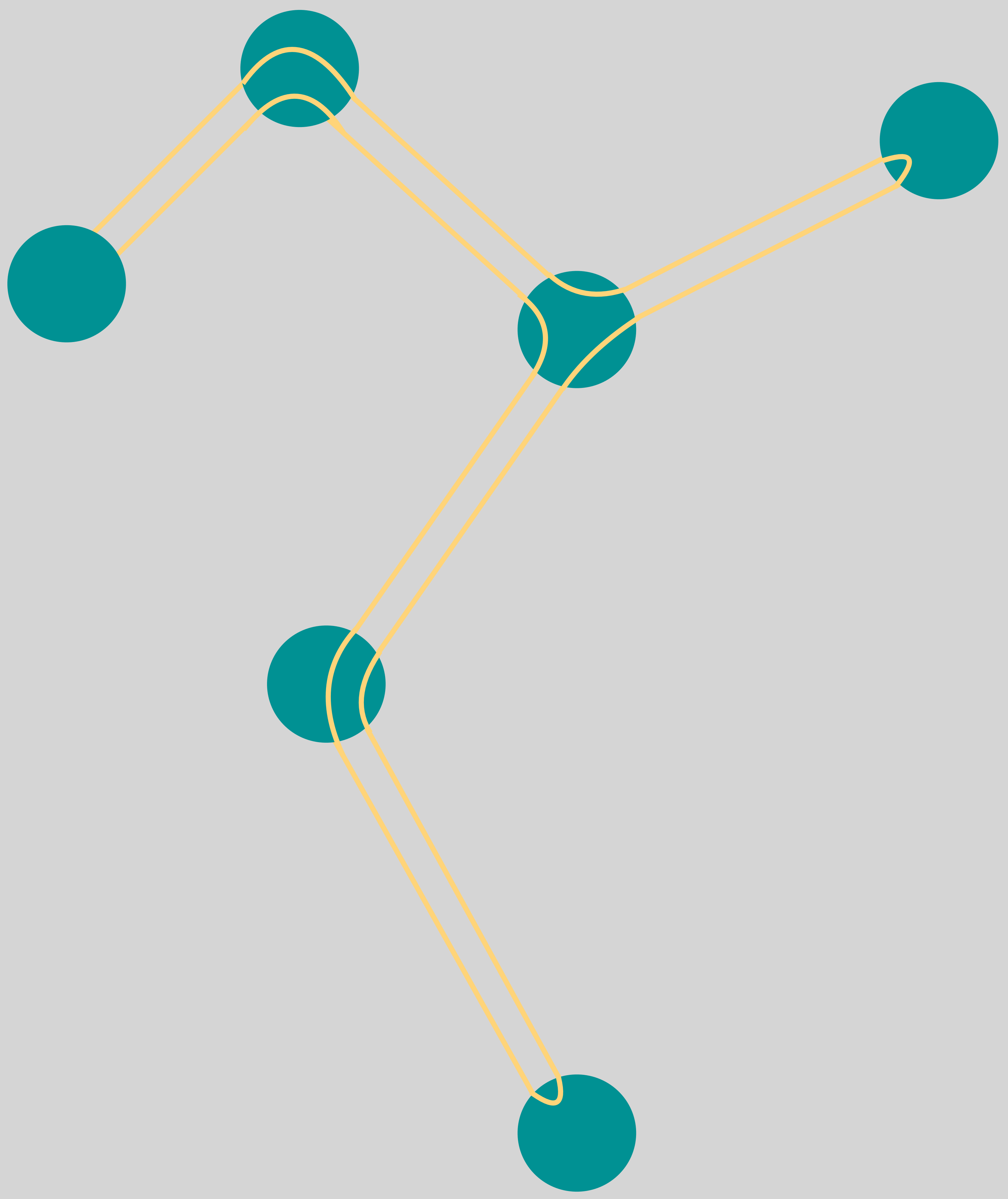
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



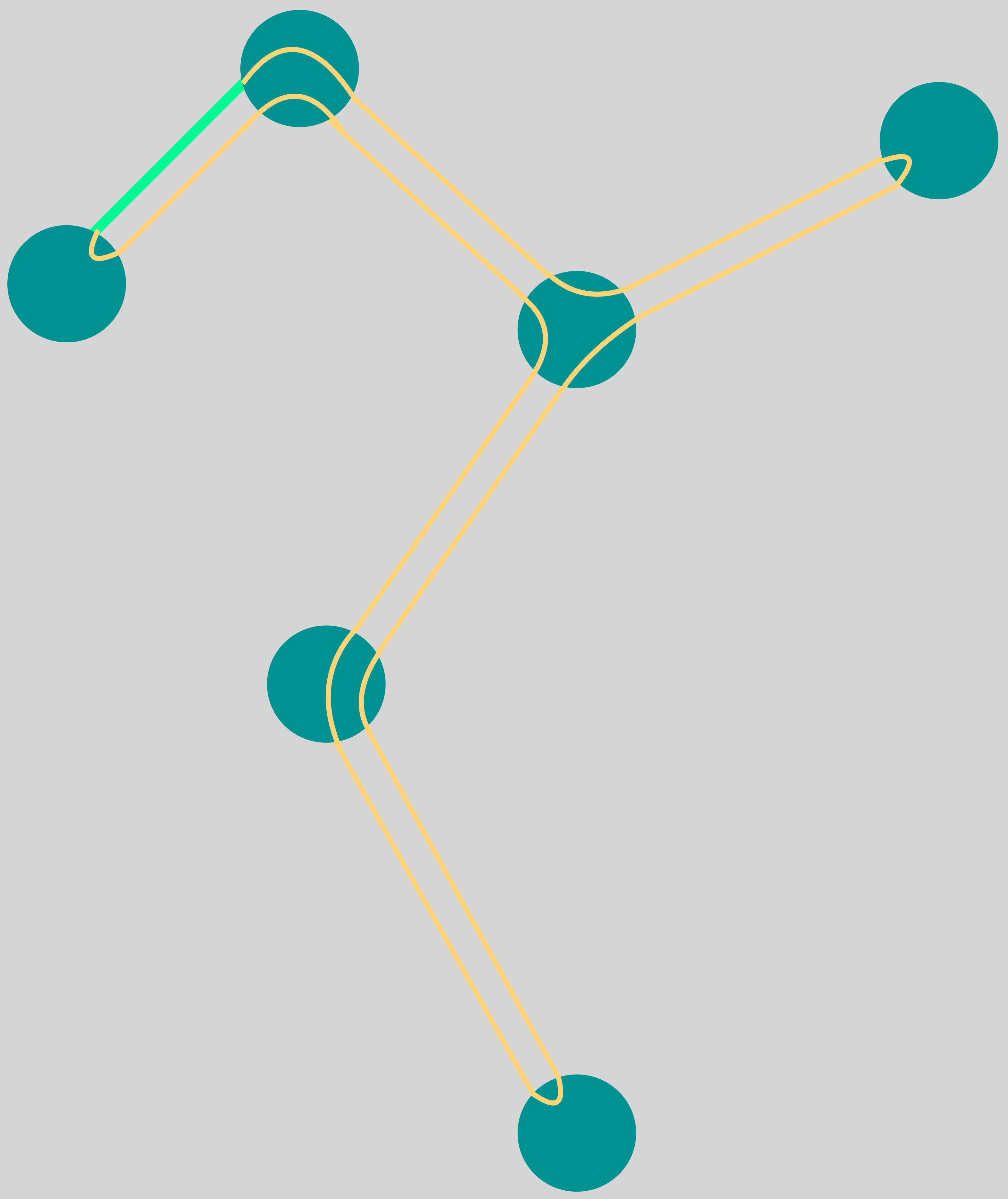
1. Find the MST T

2. Duplicate all edges of T

3. Find Eulerian Tour W

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

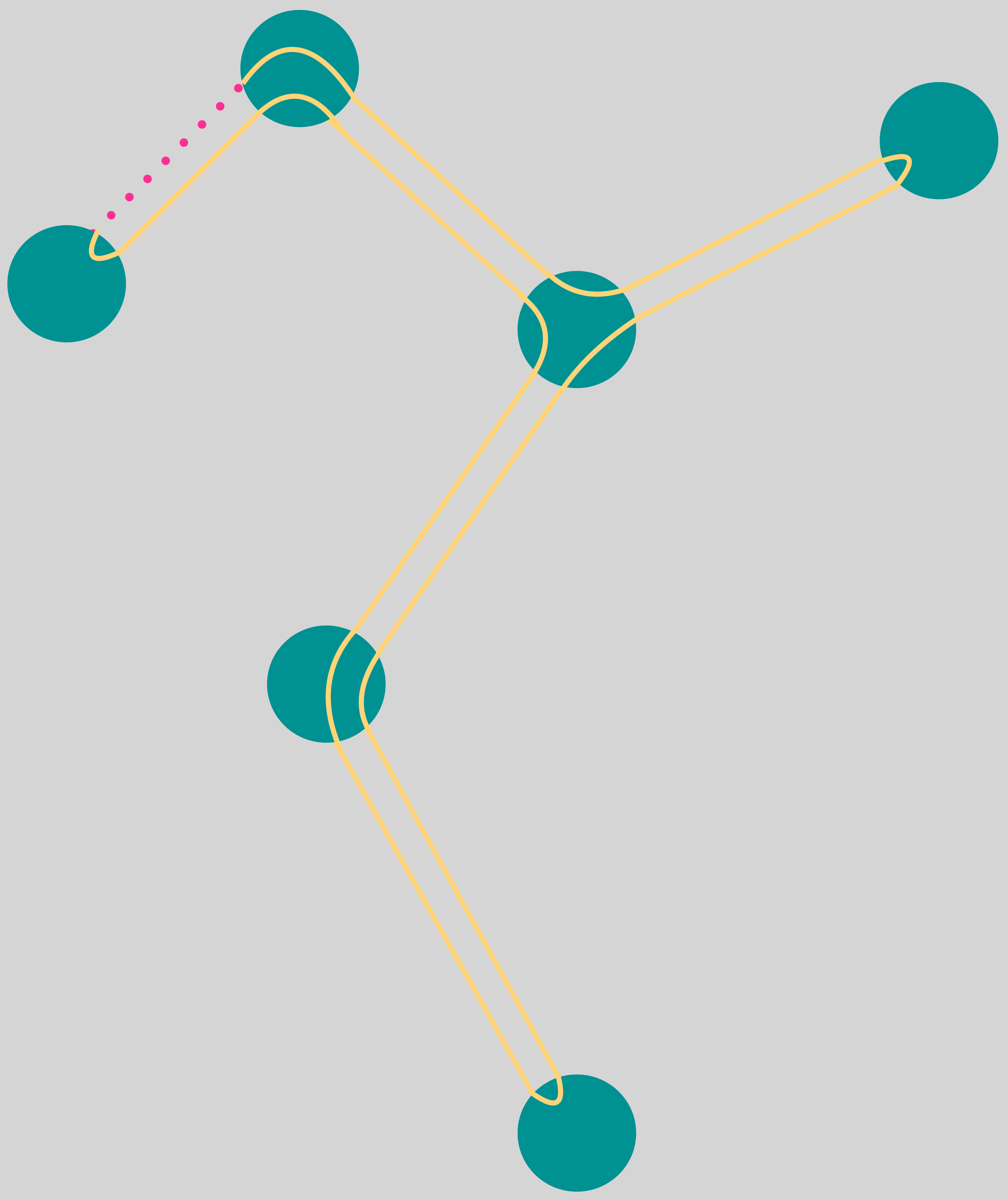
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

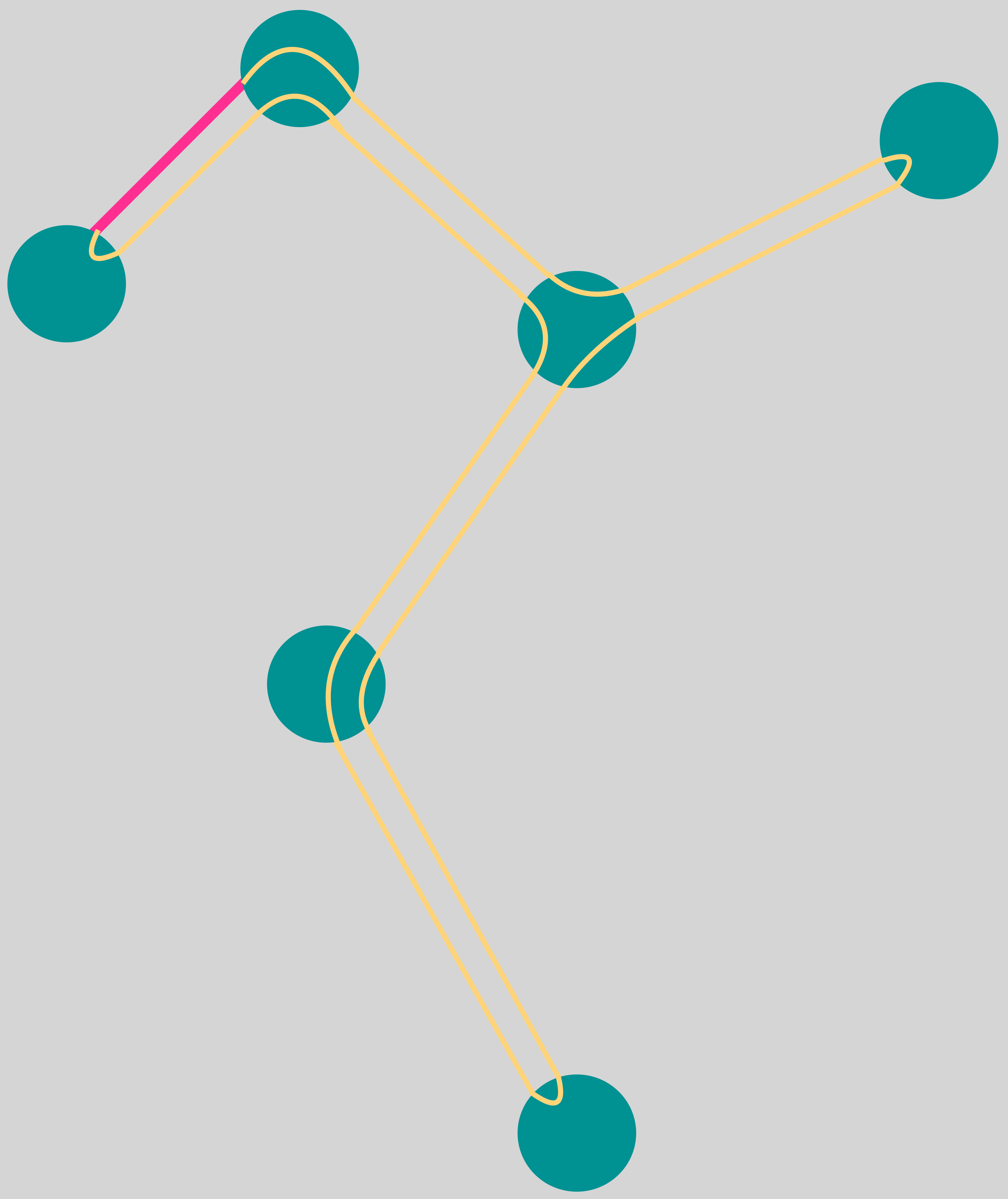
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

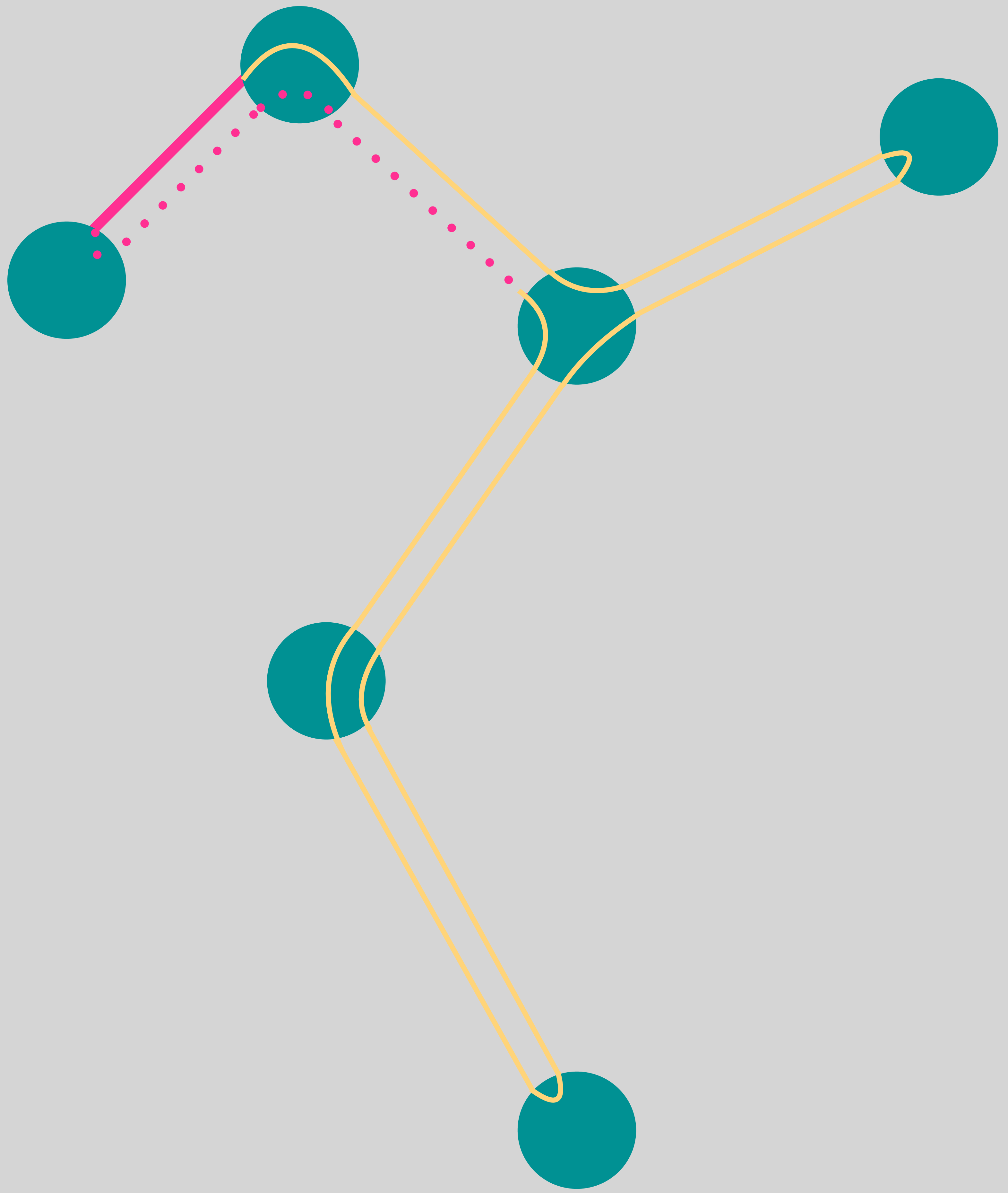
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

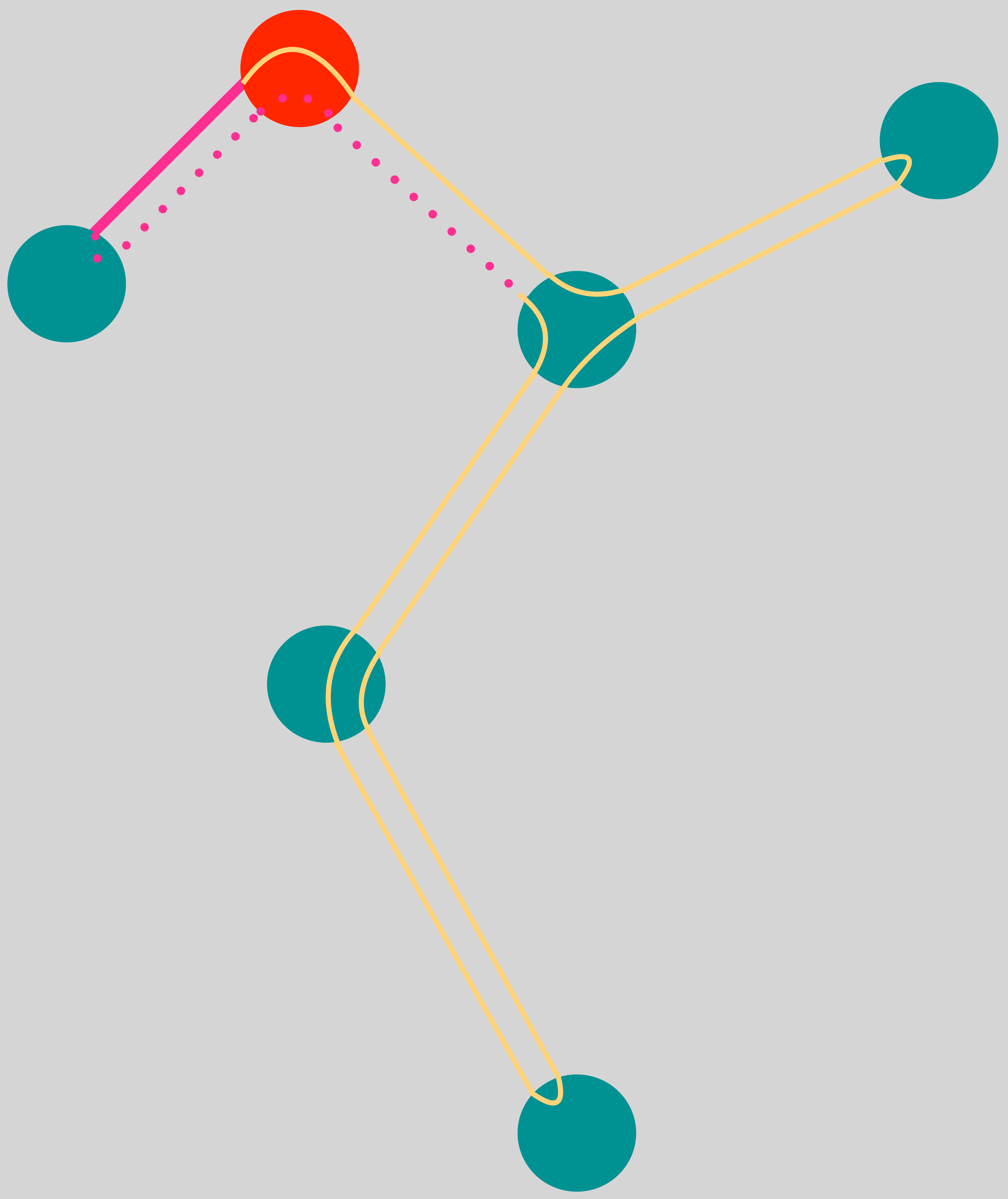
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

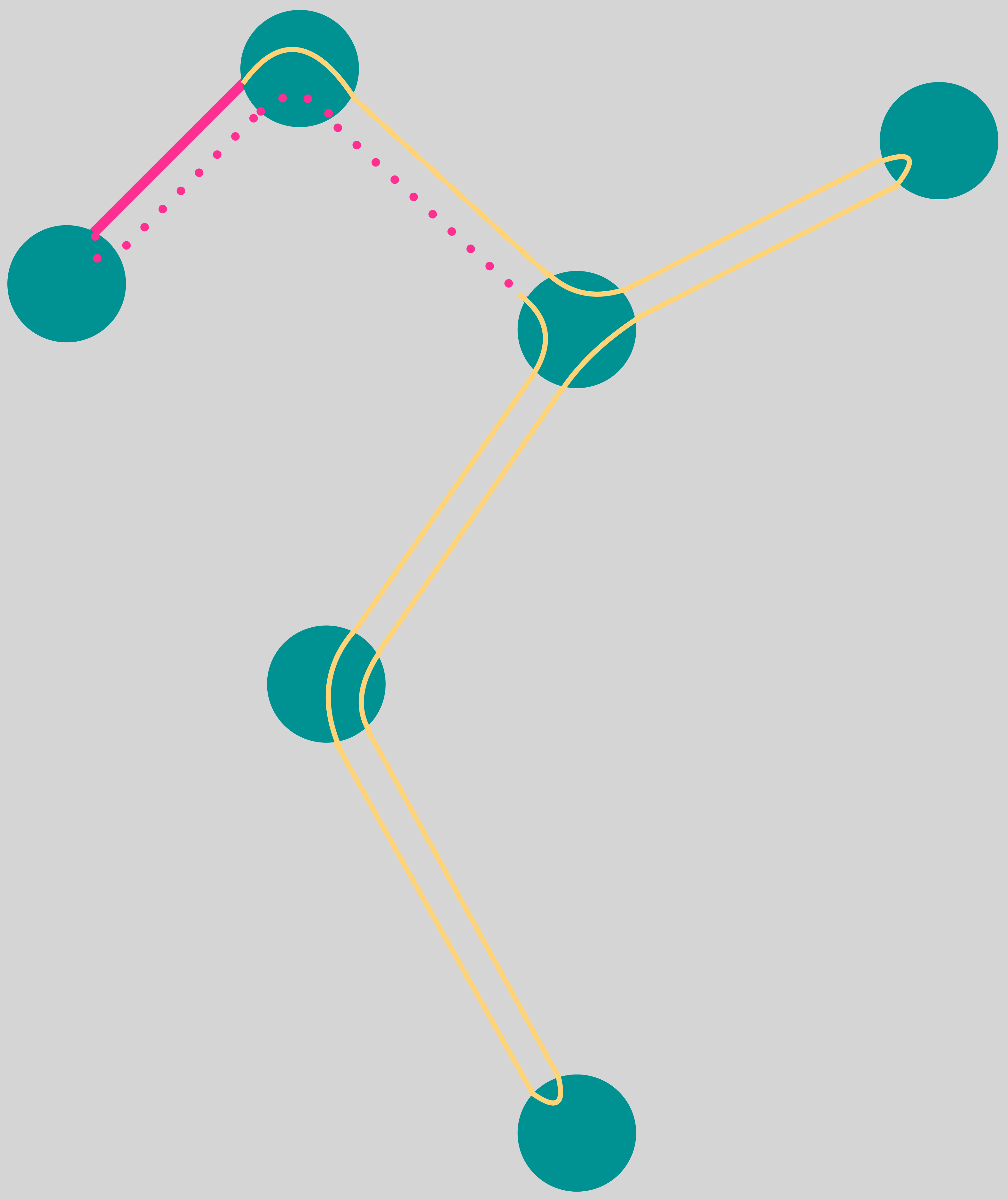
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

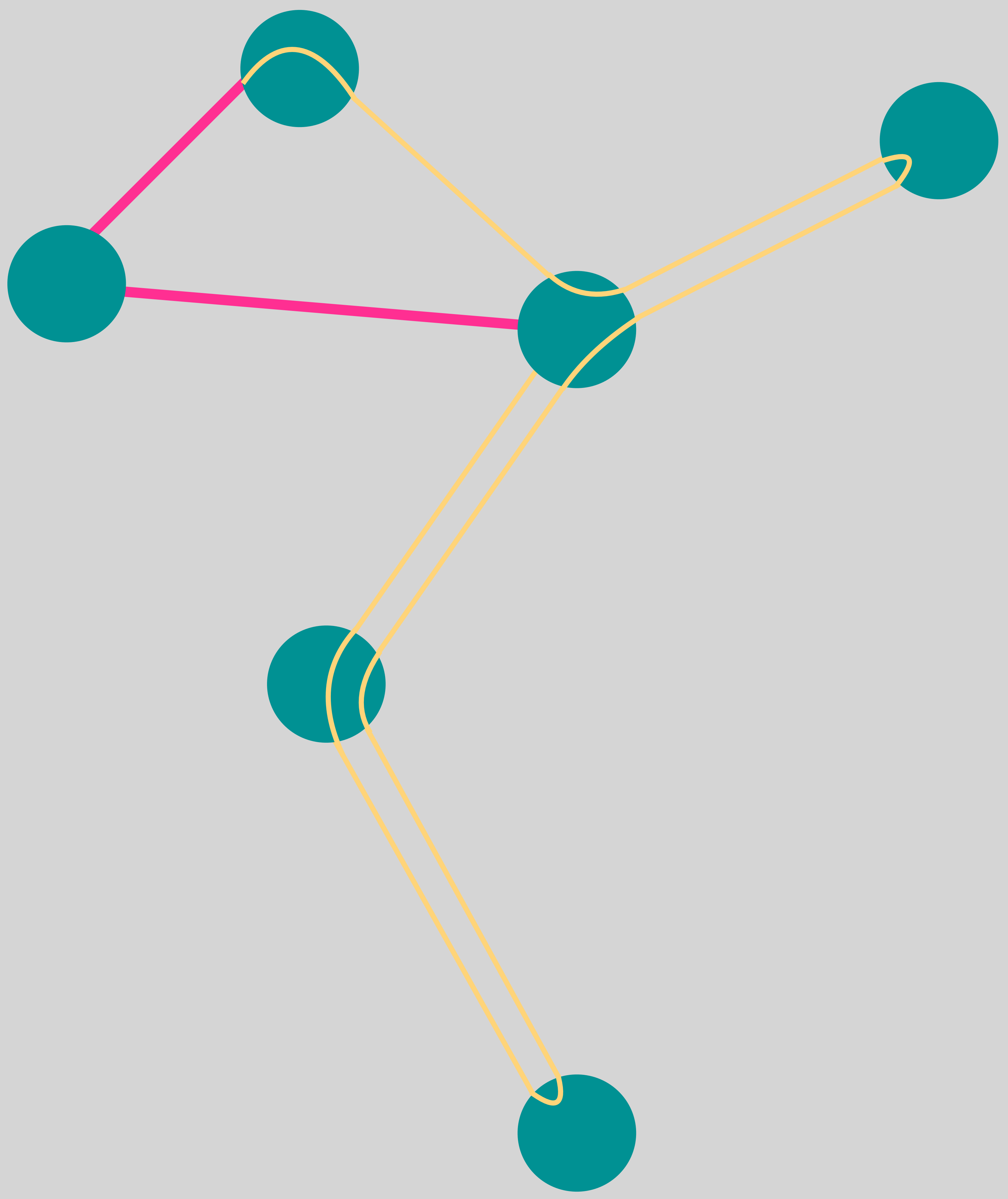
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

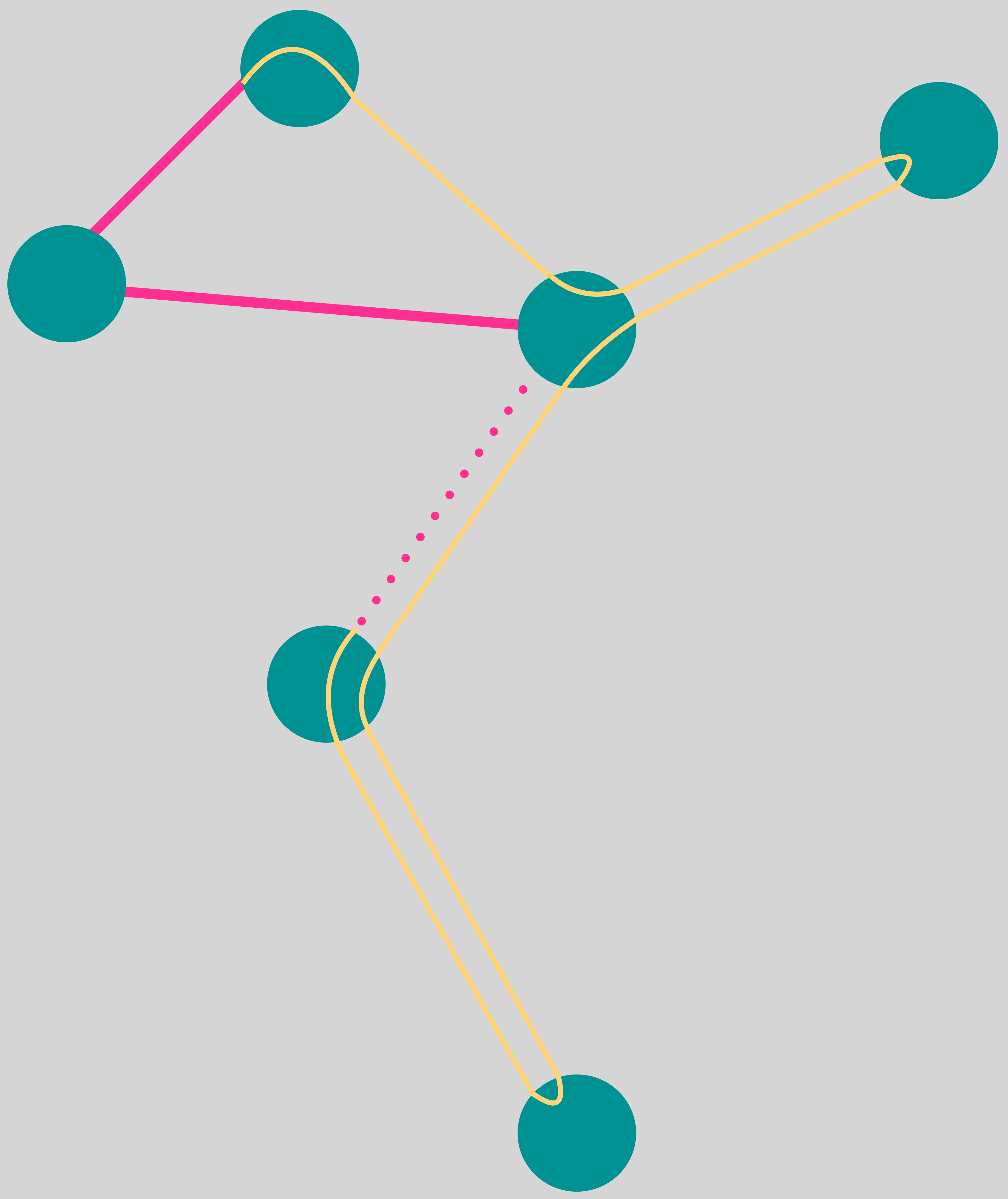
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

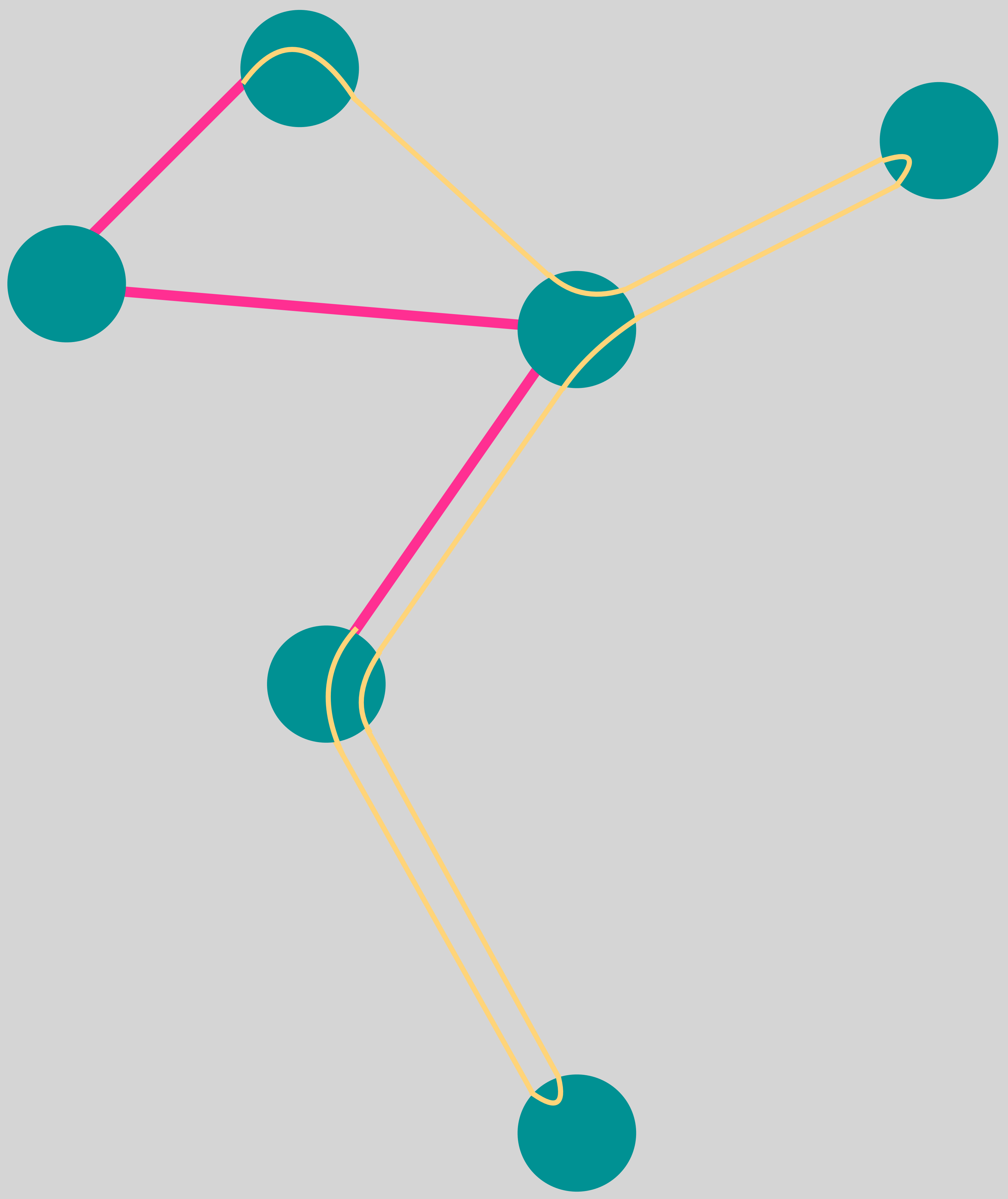
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

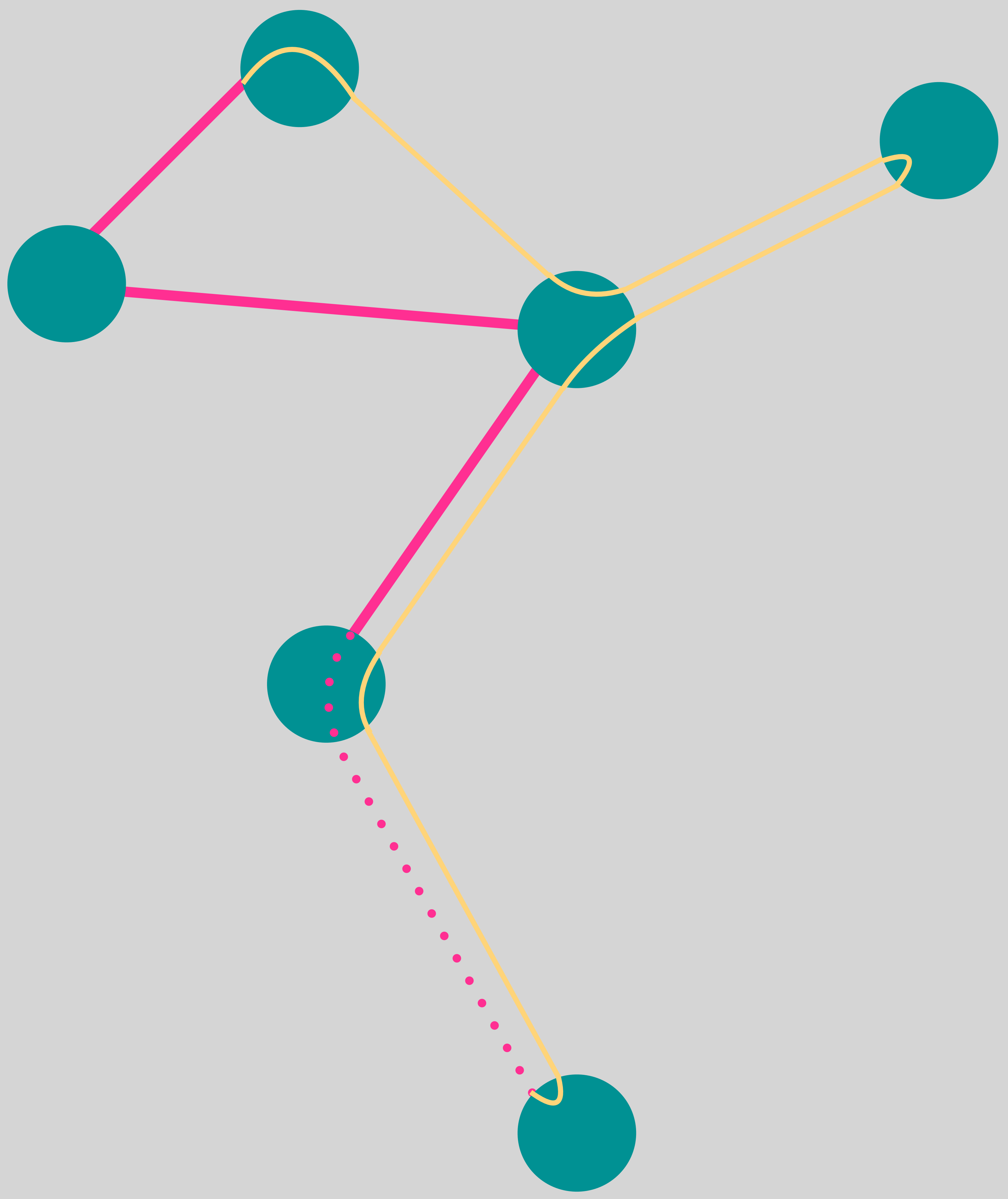
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

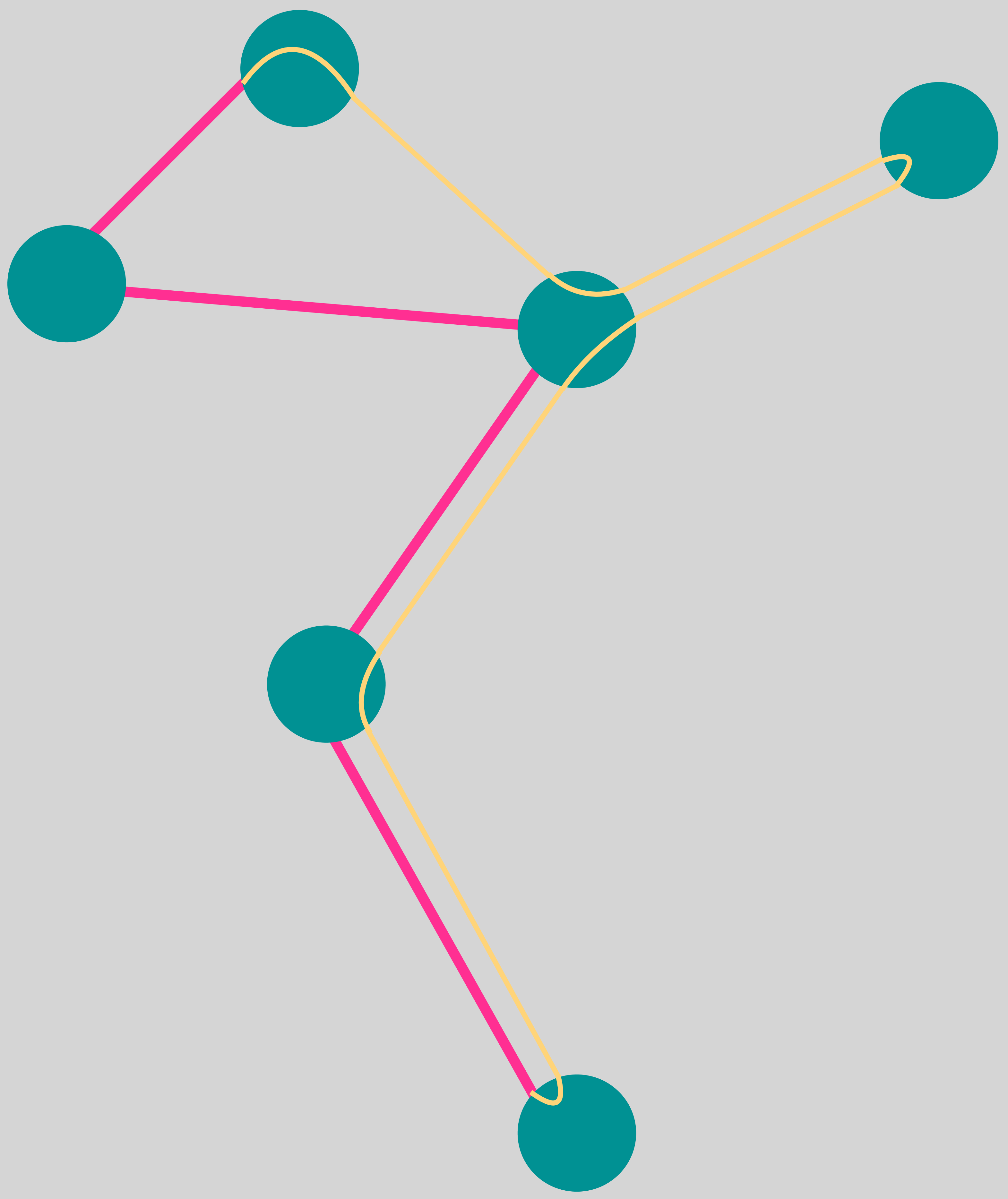
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

⇒ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

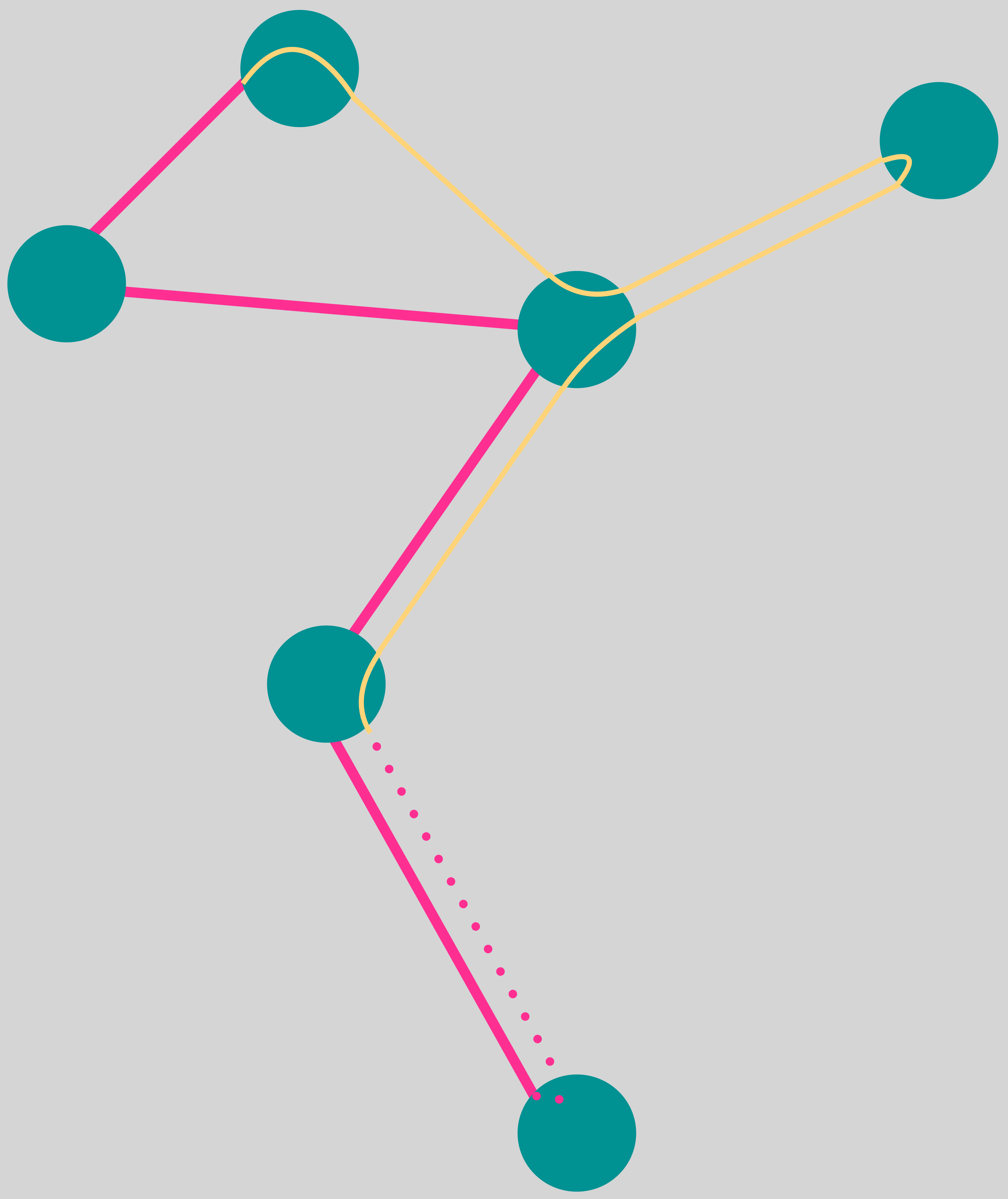
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

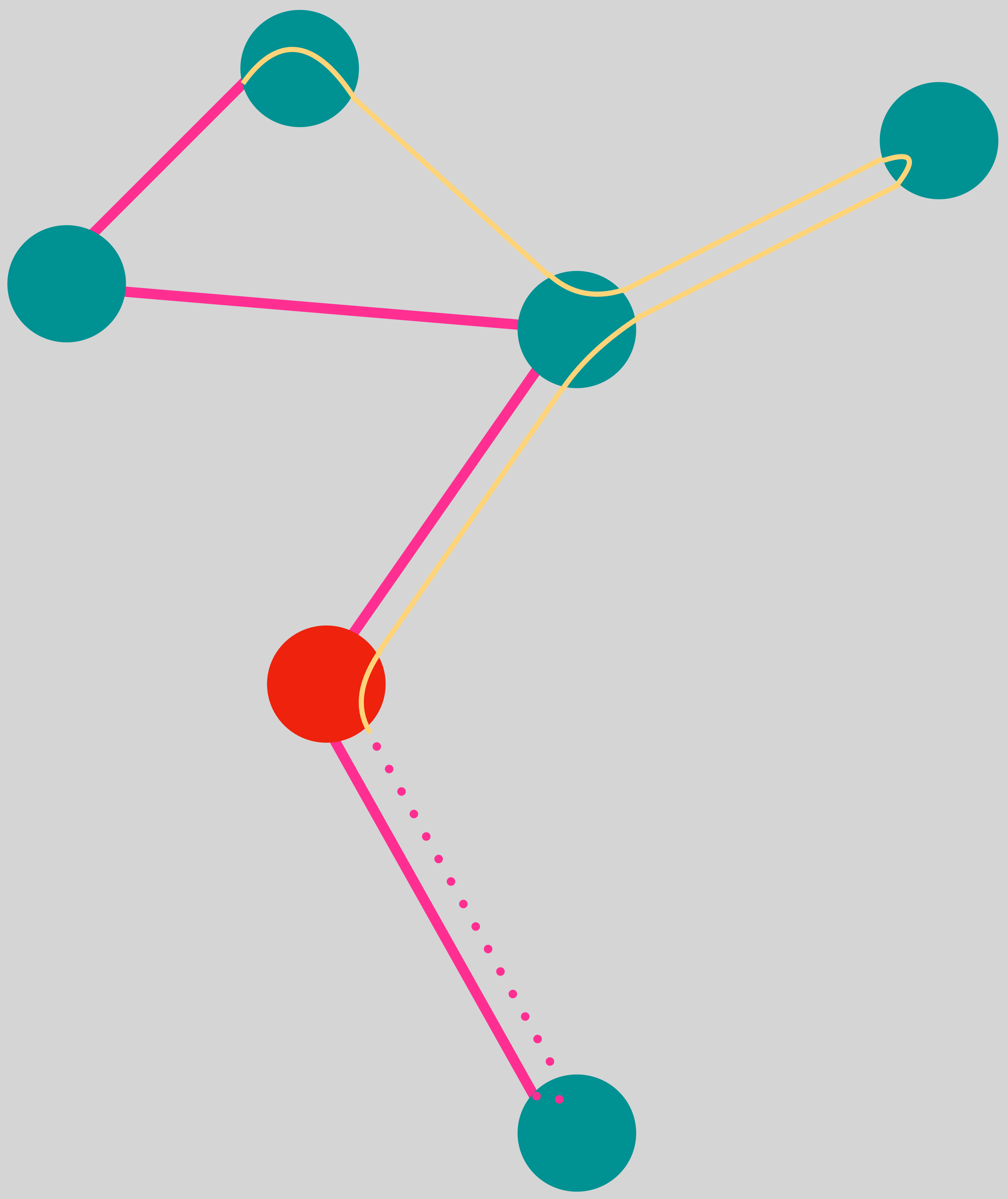
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

⇒ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

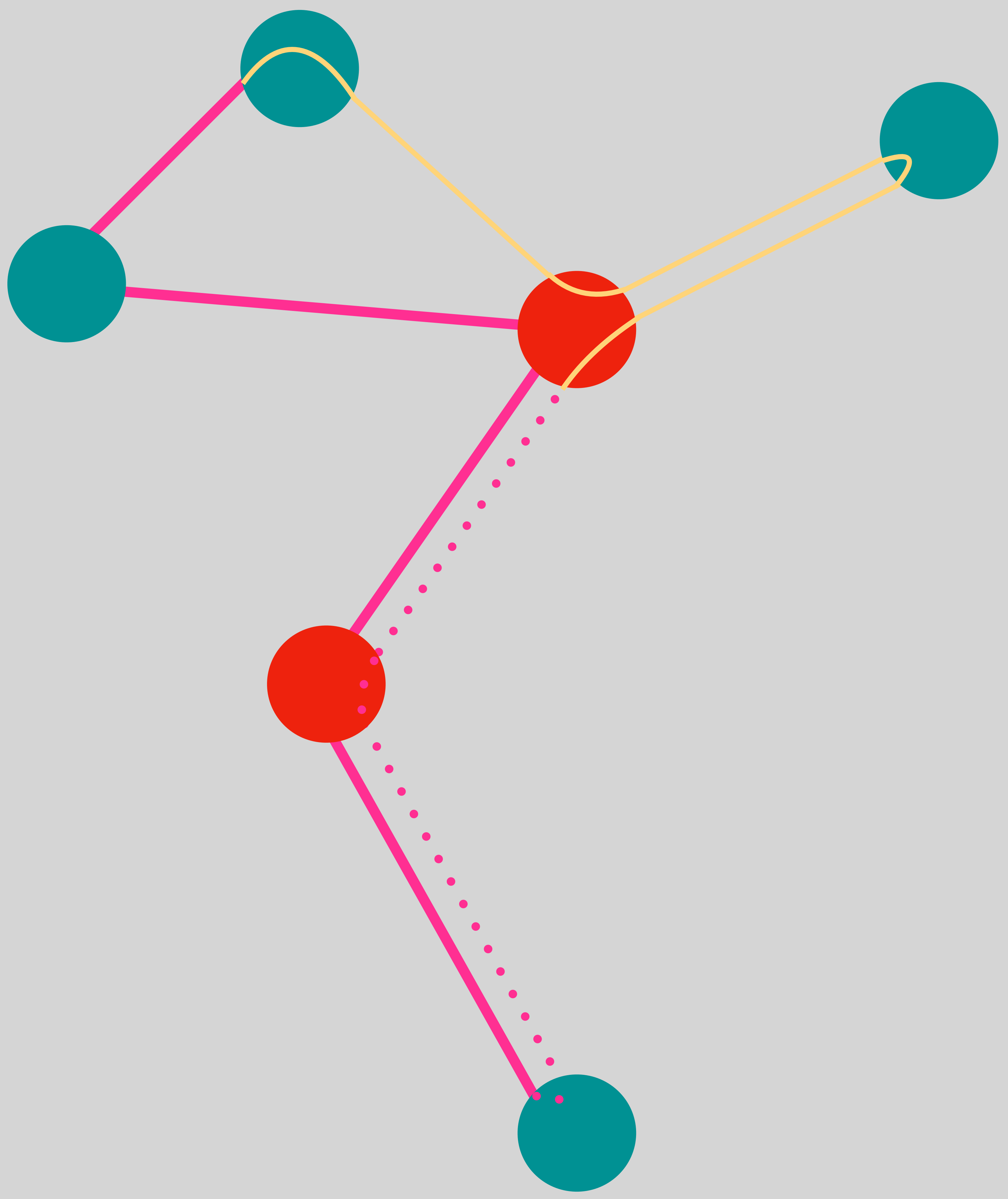
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

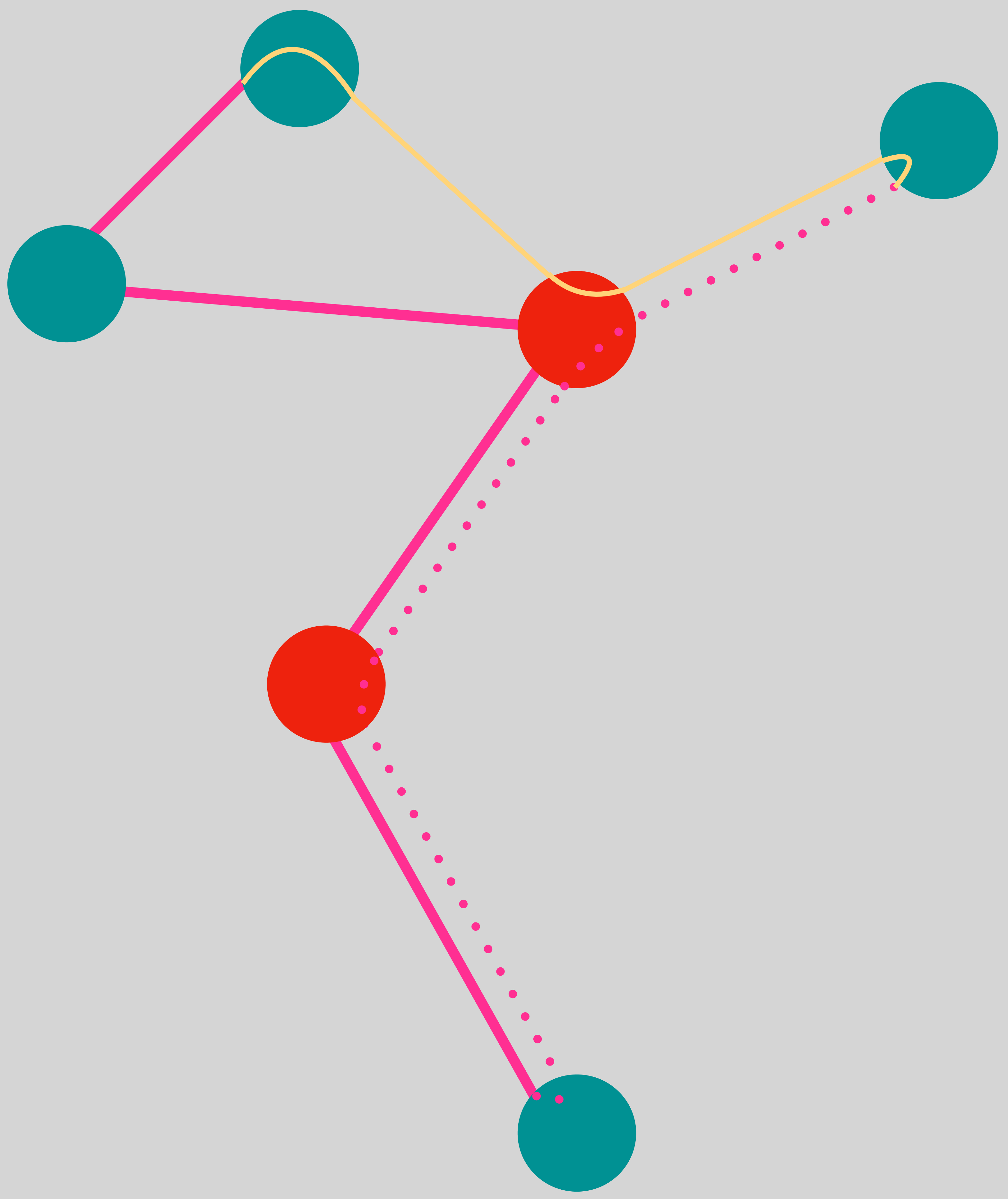
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

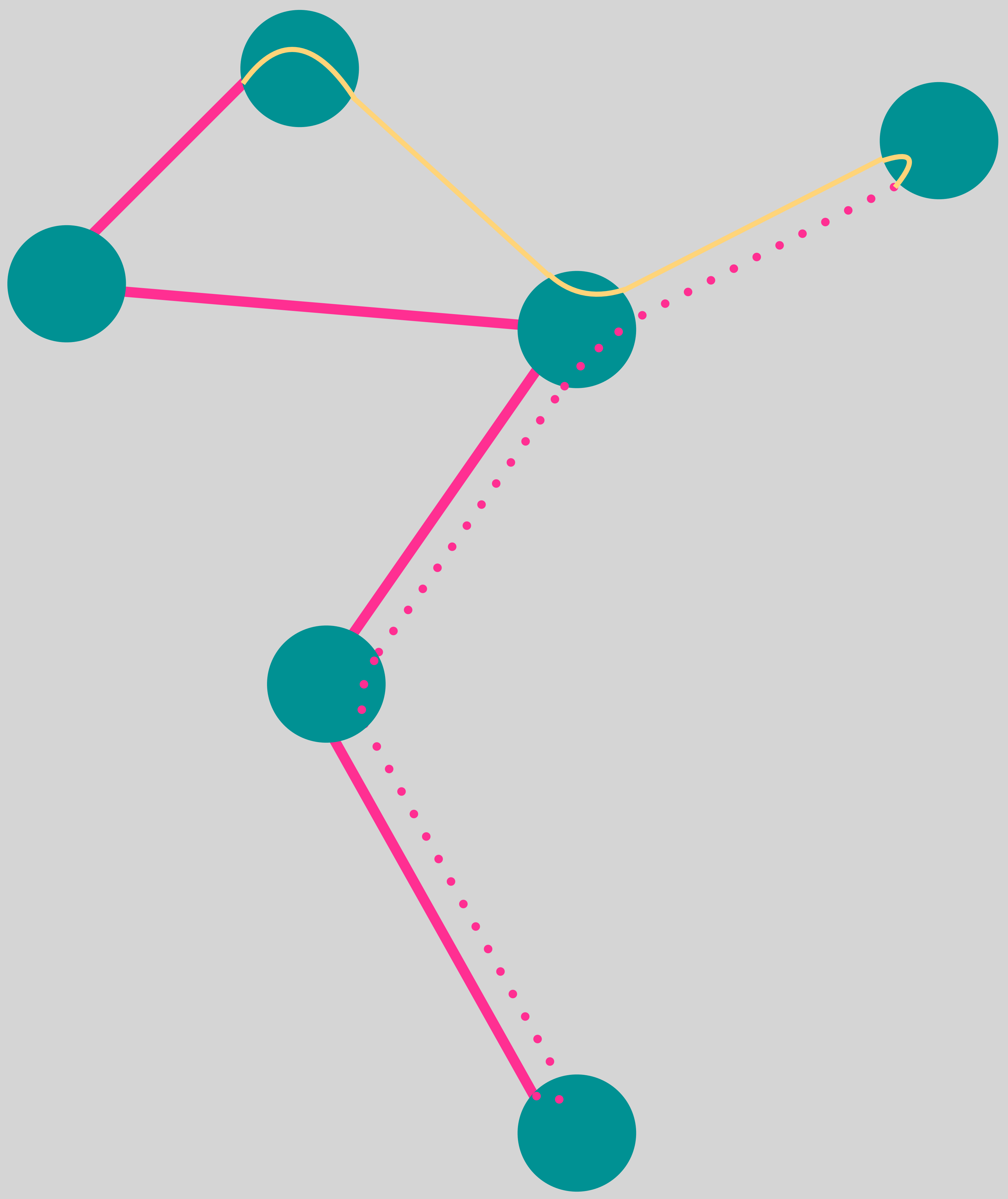
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

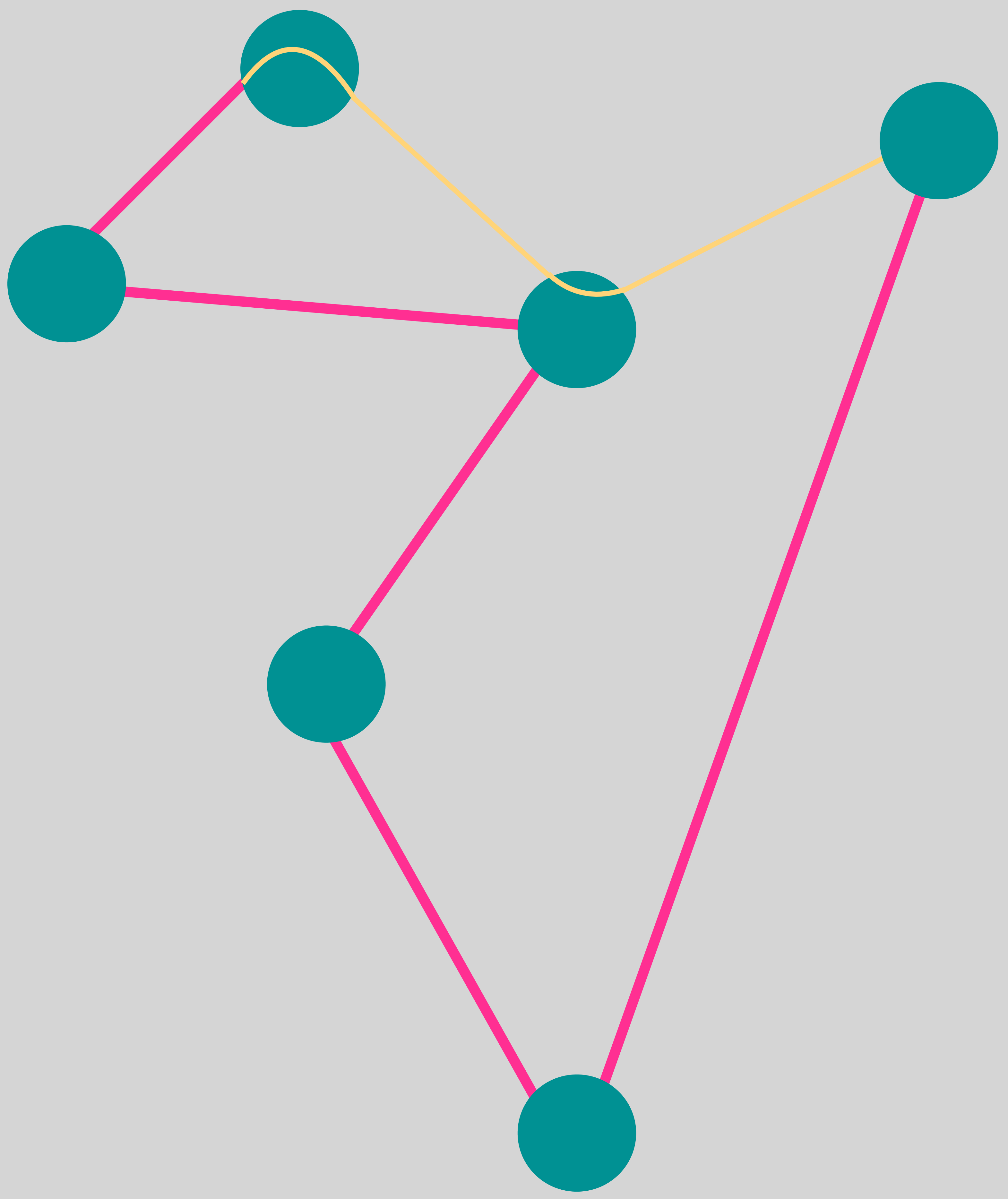
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

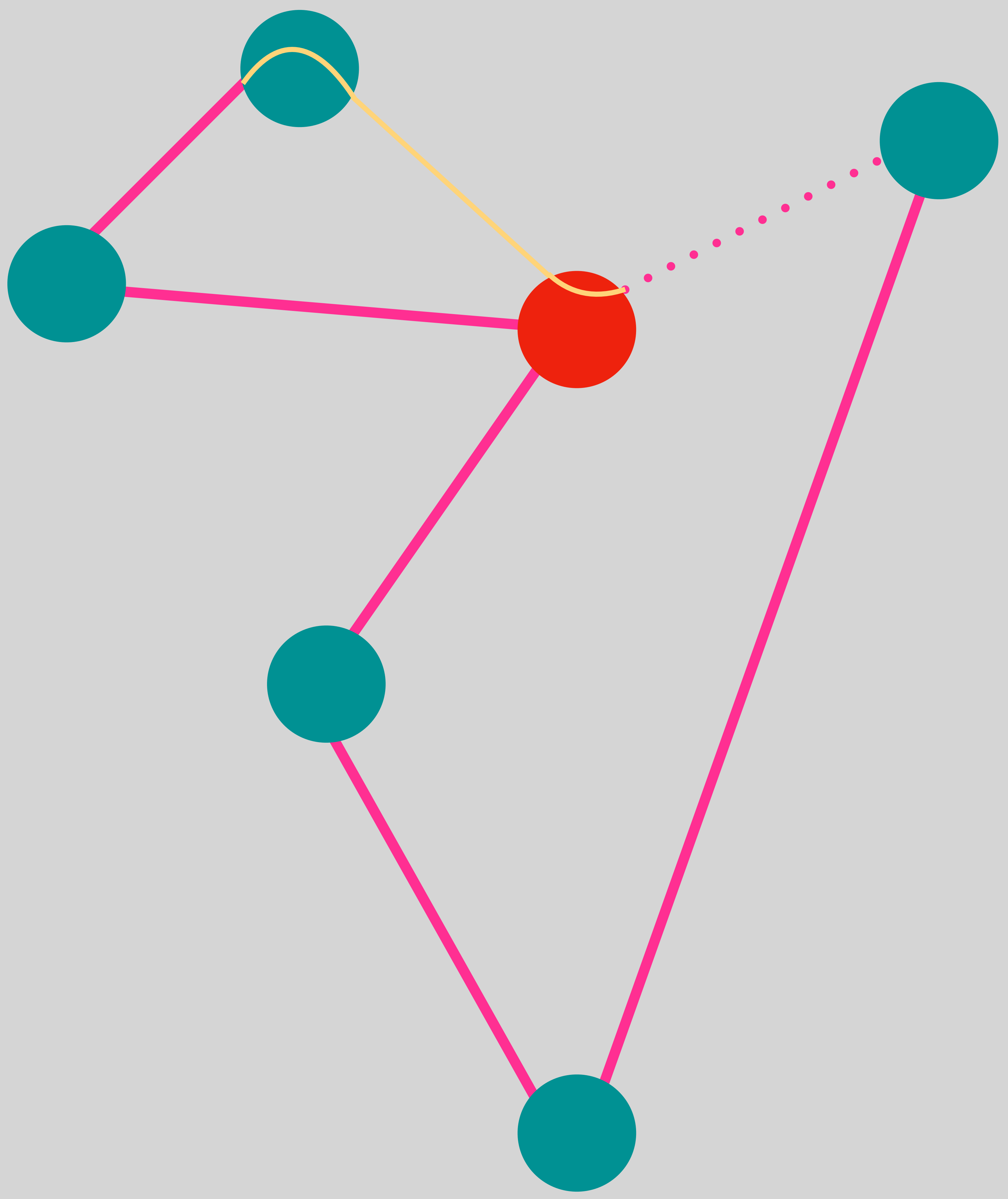
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

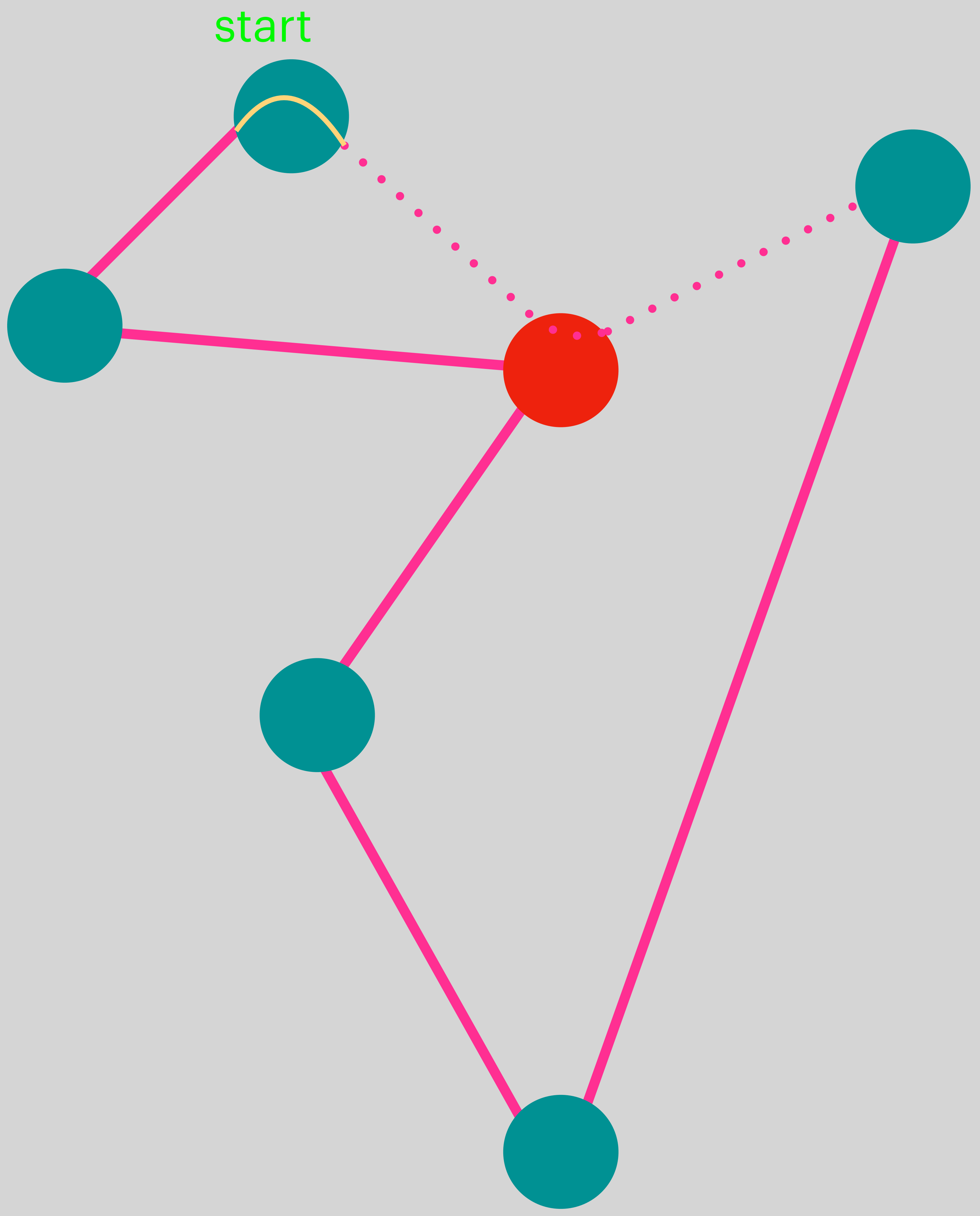
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

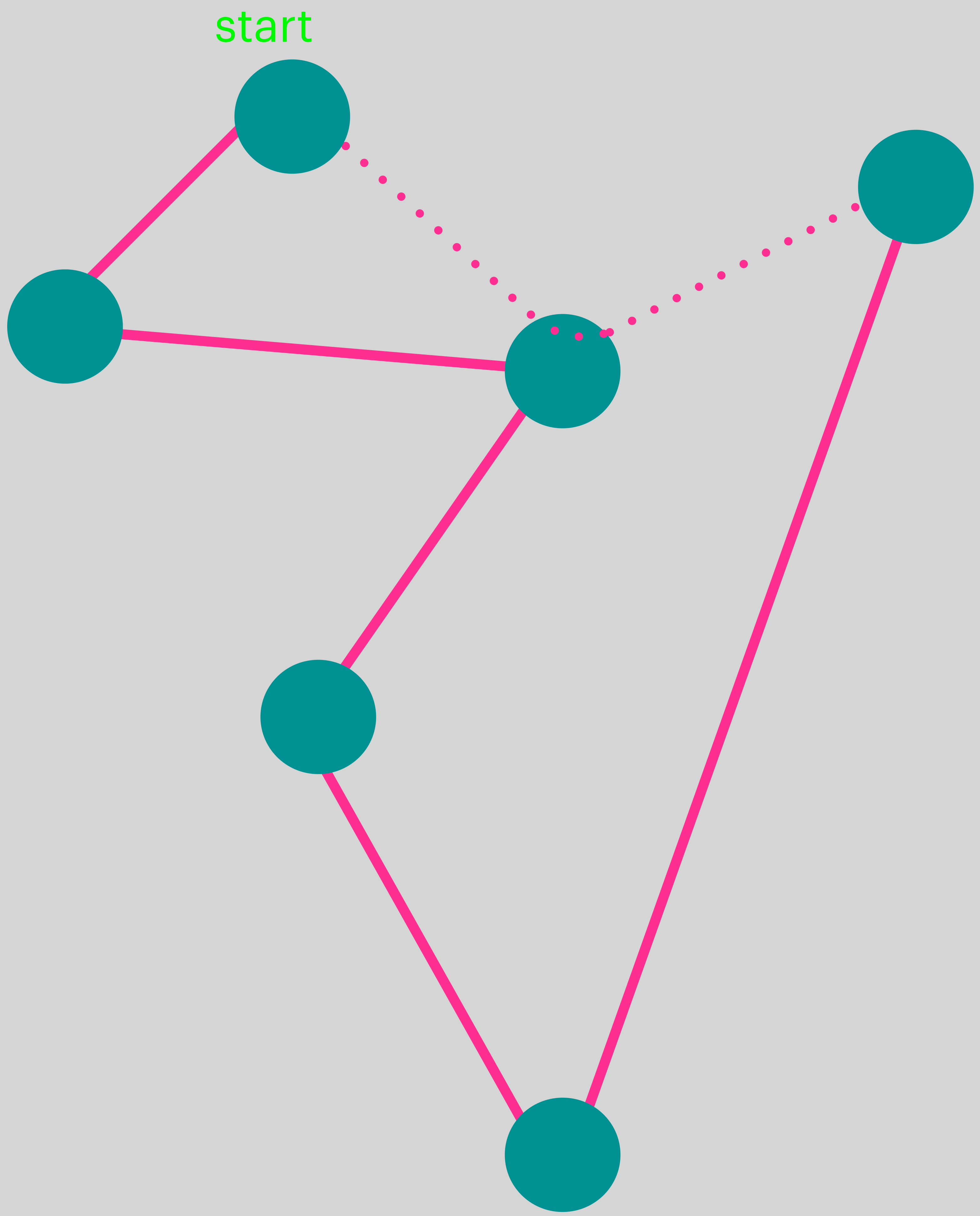
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

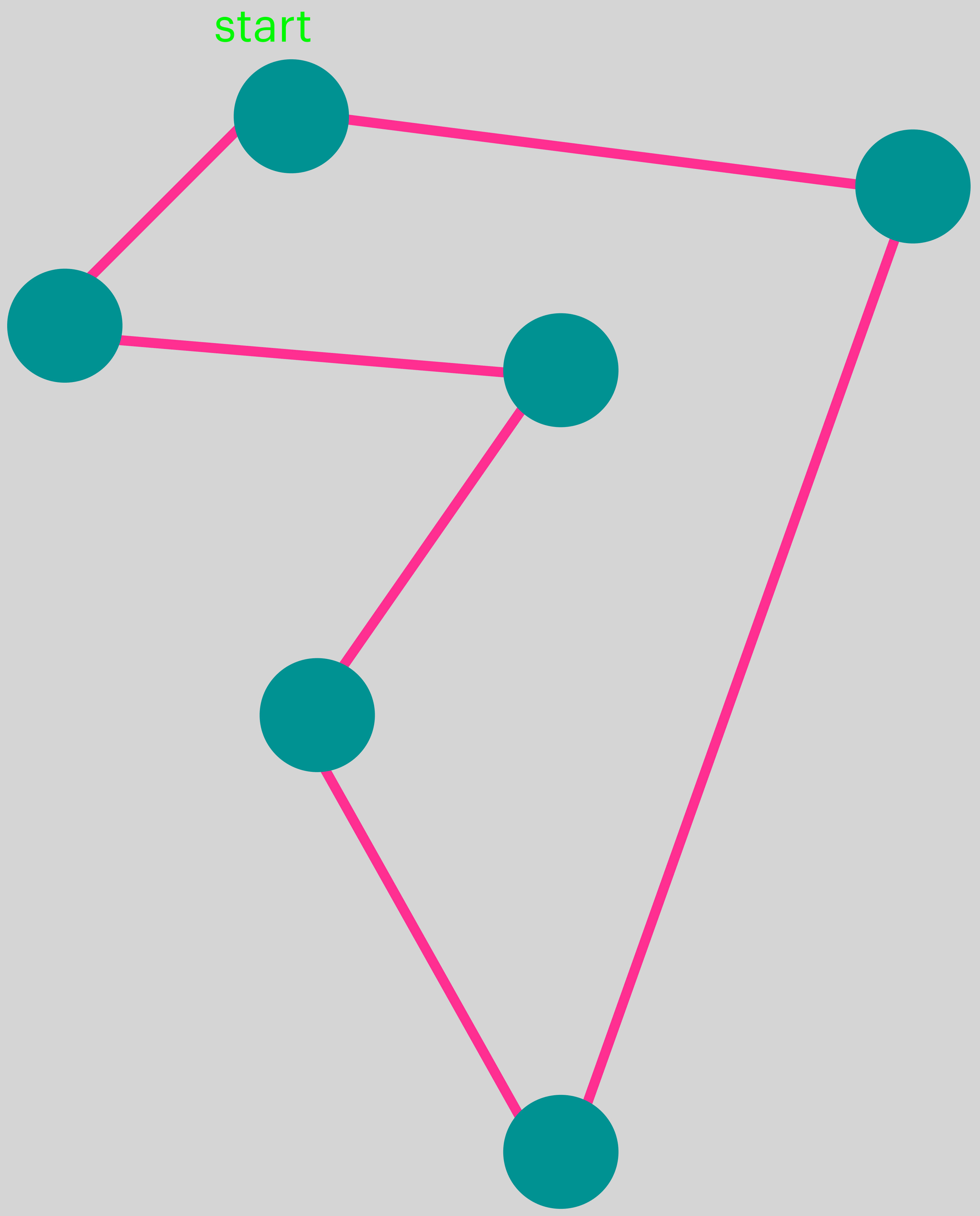
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

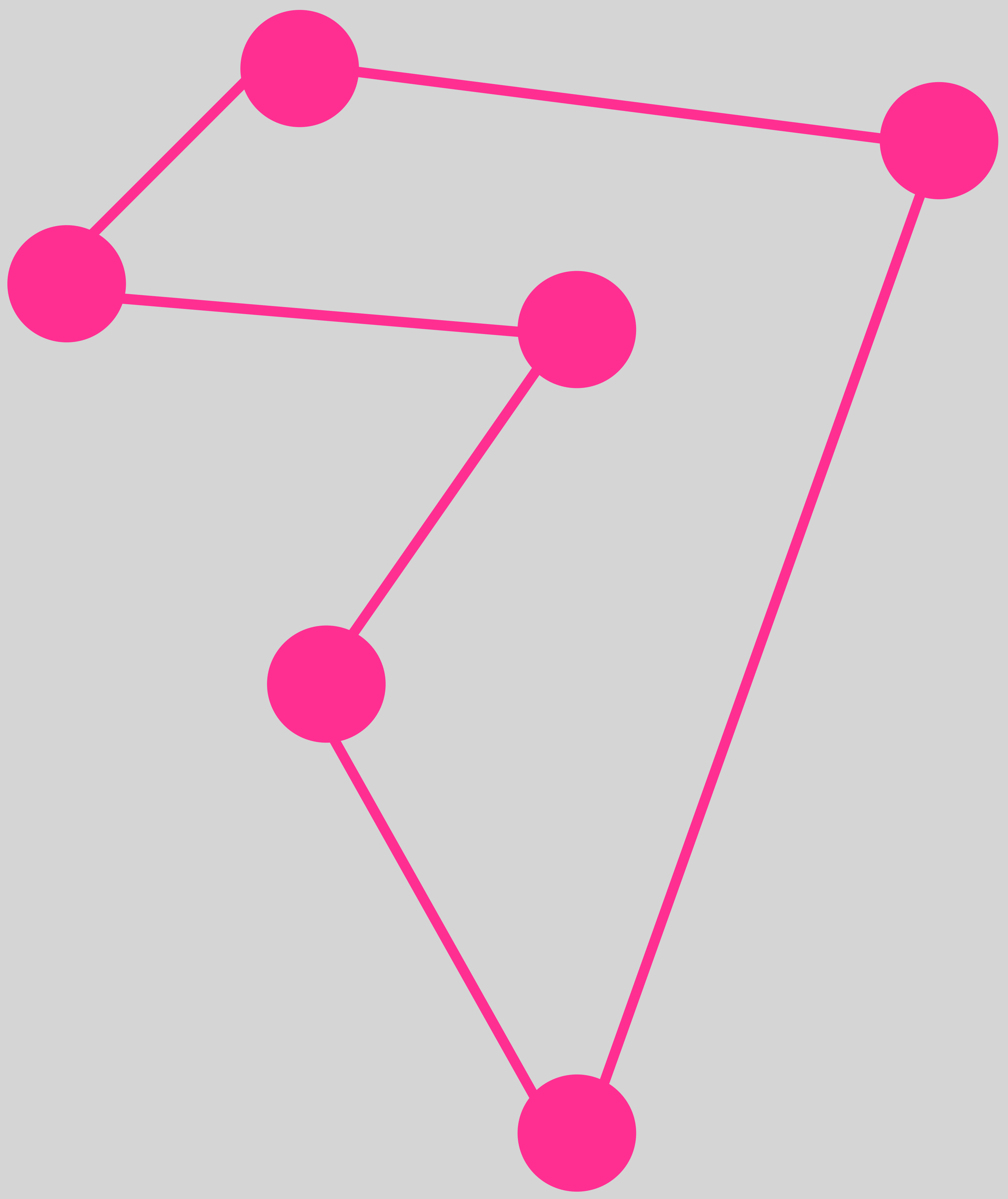
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

Metric TSP : 2-Approximation

Algorithm



1. Find the MST T

2. Duplicate all edges of T

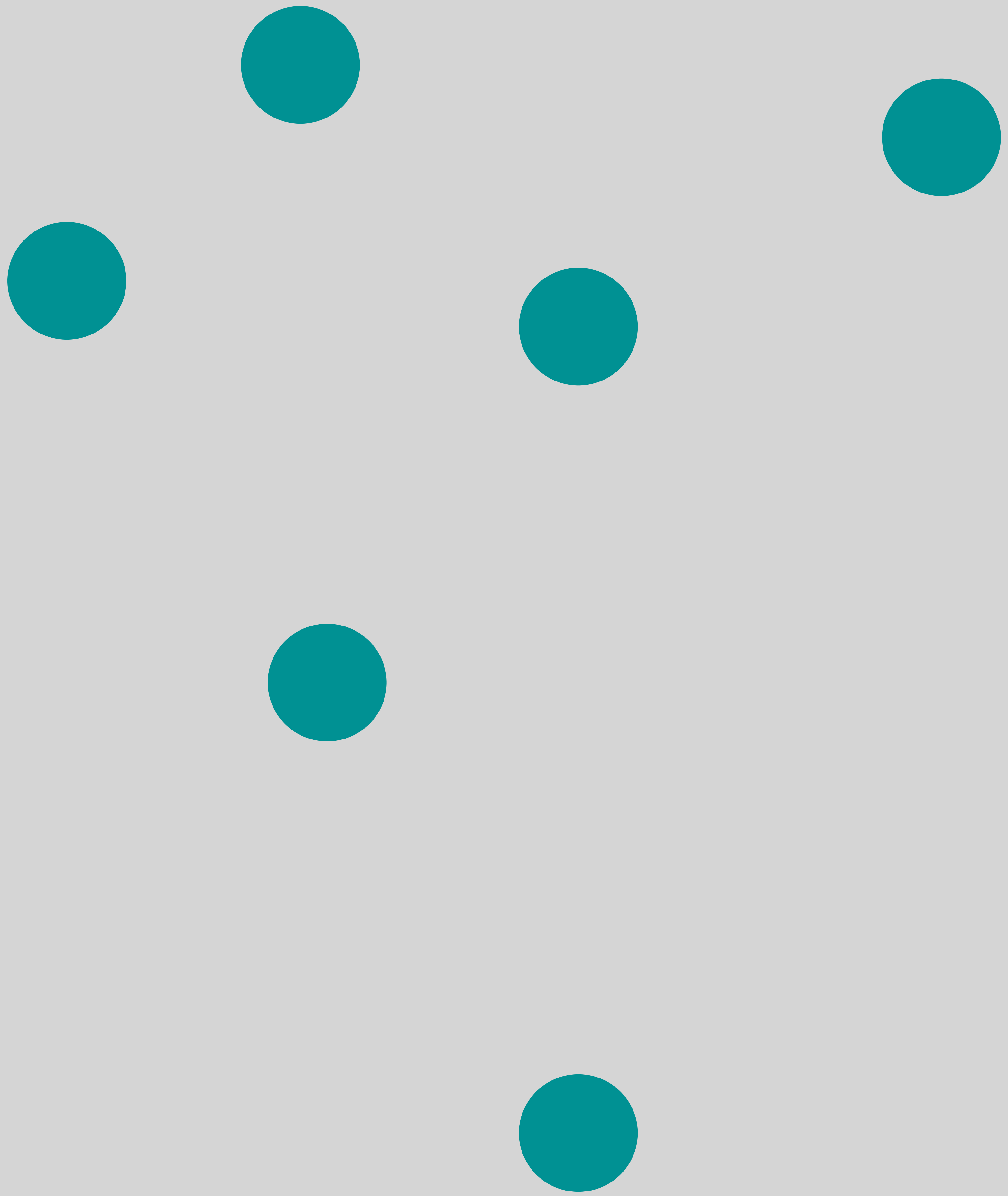
3. Find Eulerian Tour W

4. Traverse W once using shortcuts s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

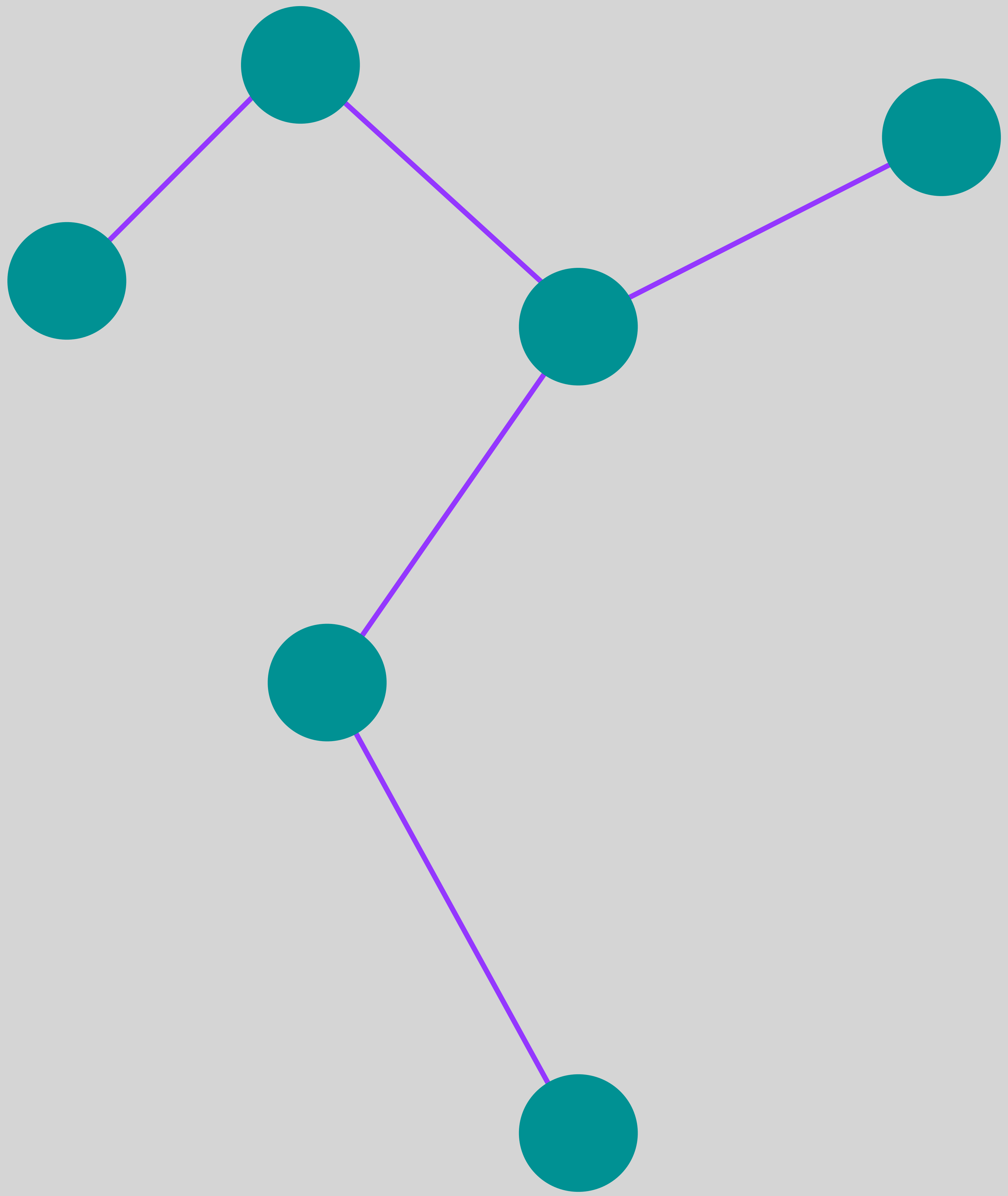
Metric TSP : 2-Approximation

Correctness



Metric TSP : 2-Approximation

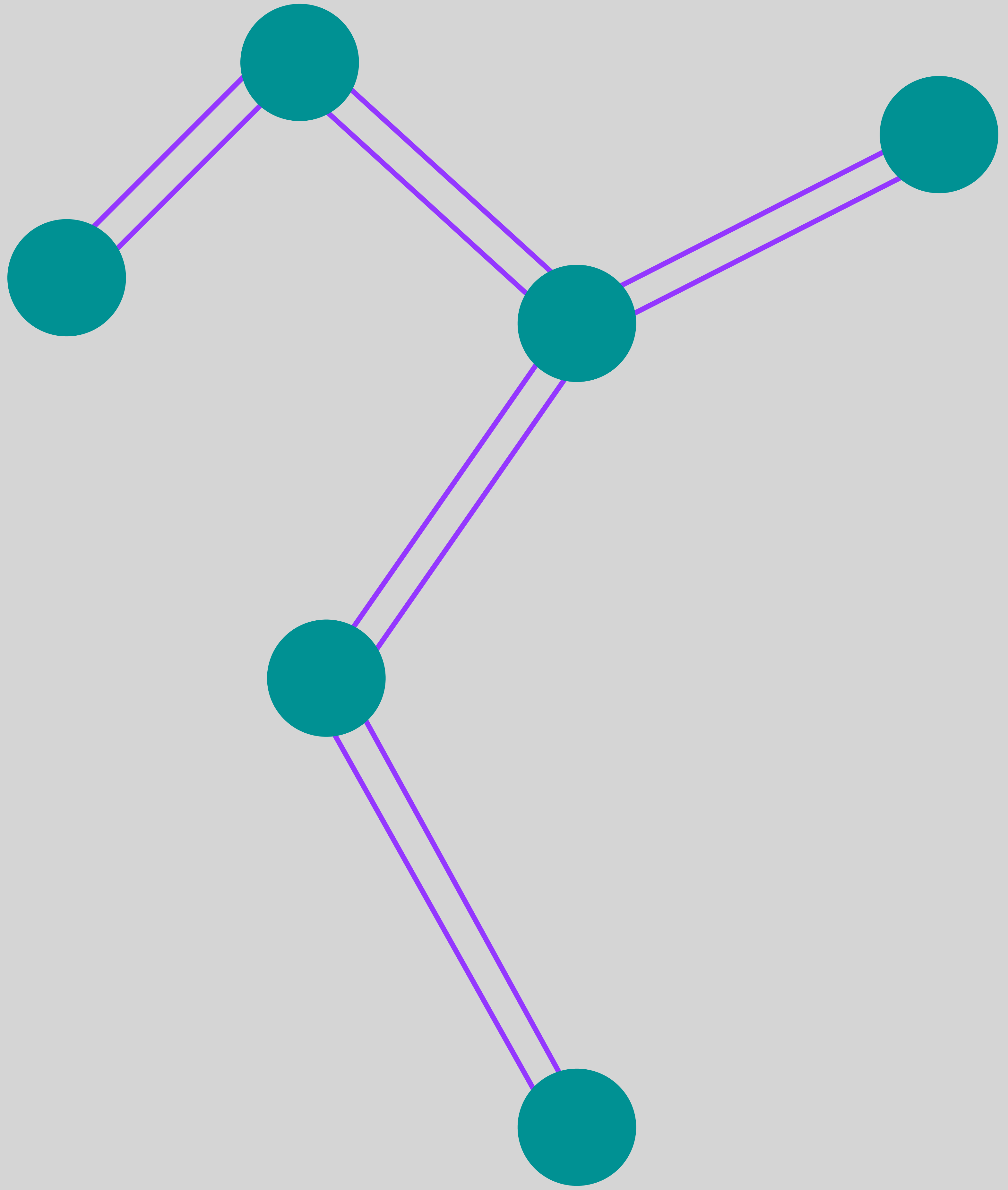
Correctness



1. Find the MST T $l(T) \leq OPT(K_n, l)$

Metric TSP : 2-Approximation

Correctness

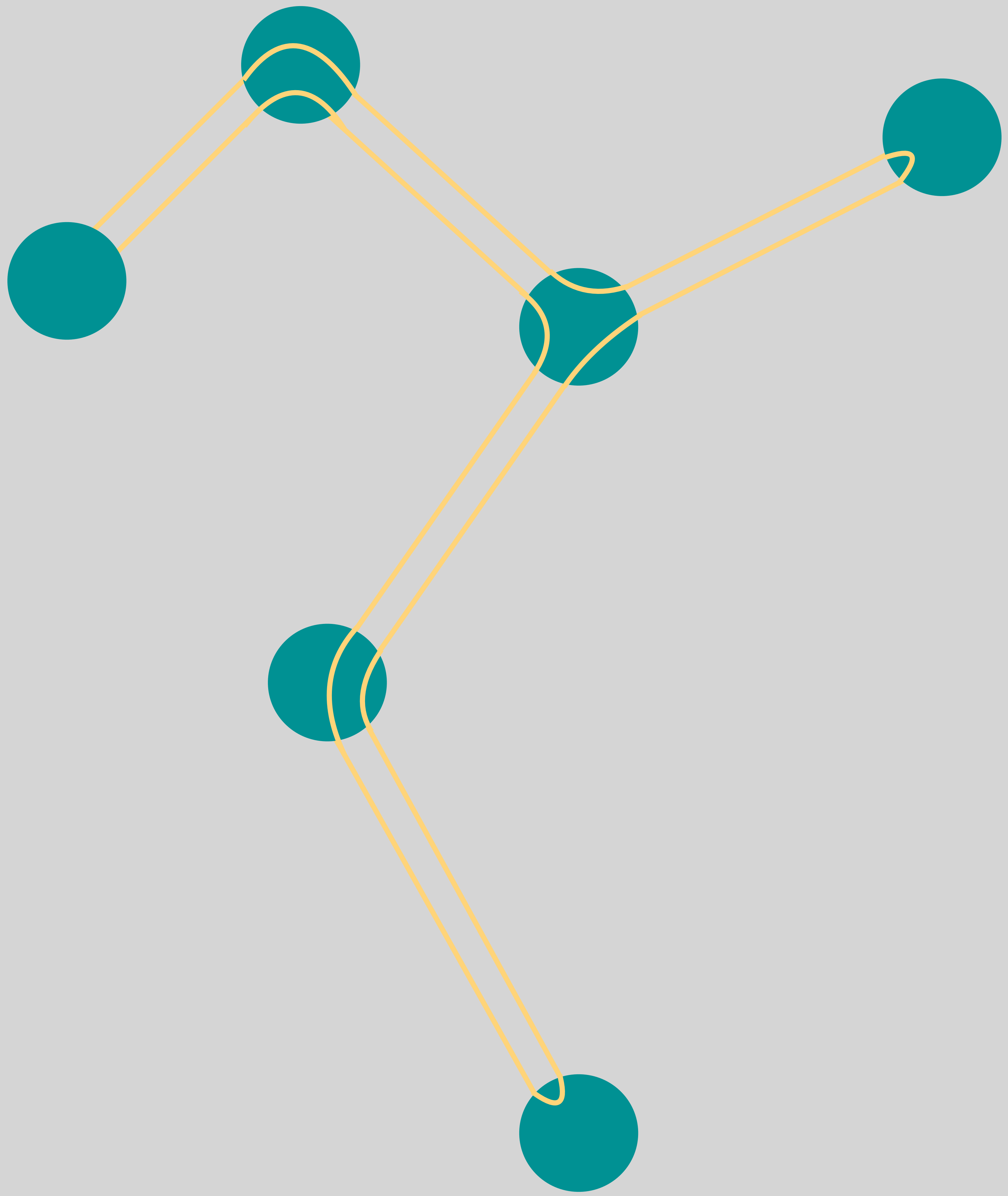


1. Find the MST T $l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $2\,l(T) \leq 2\,OPT(K_n, l)$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

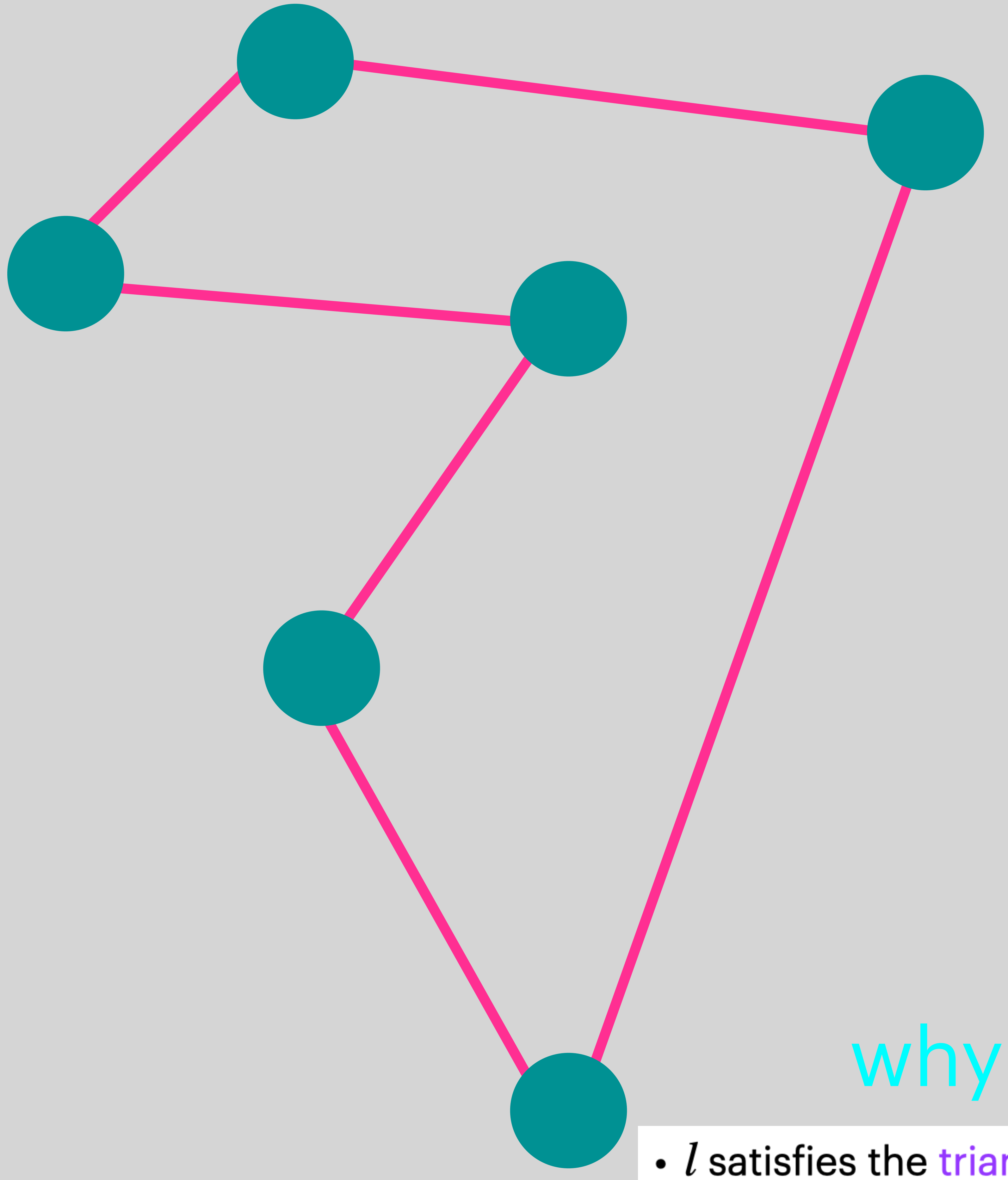
2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\qquad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\qquad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

4. Traverse W once using shortcuts
s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

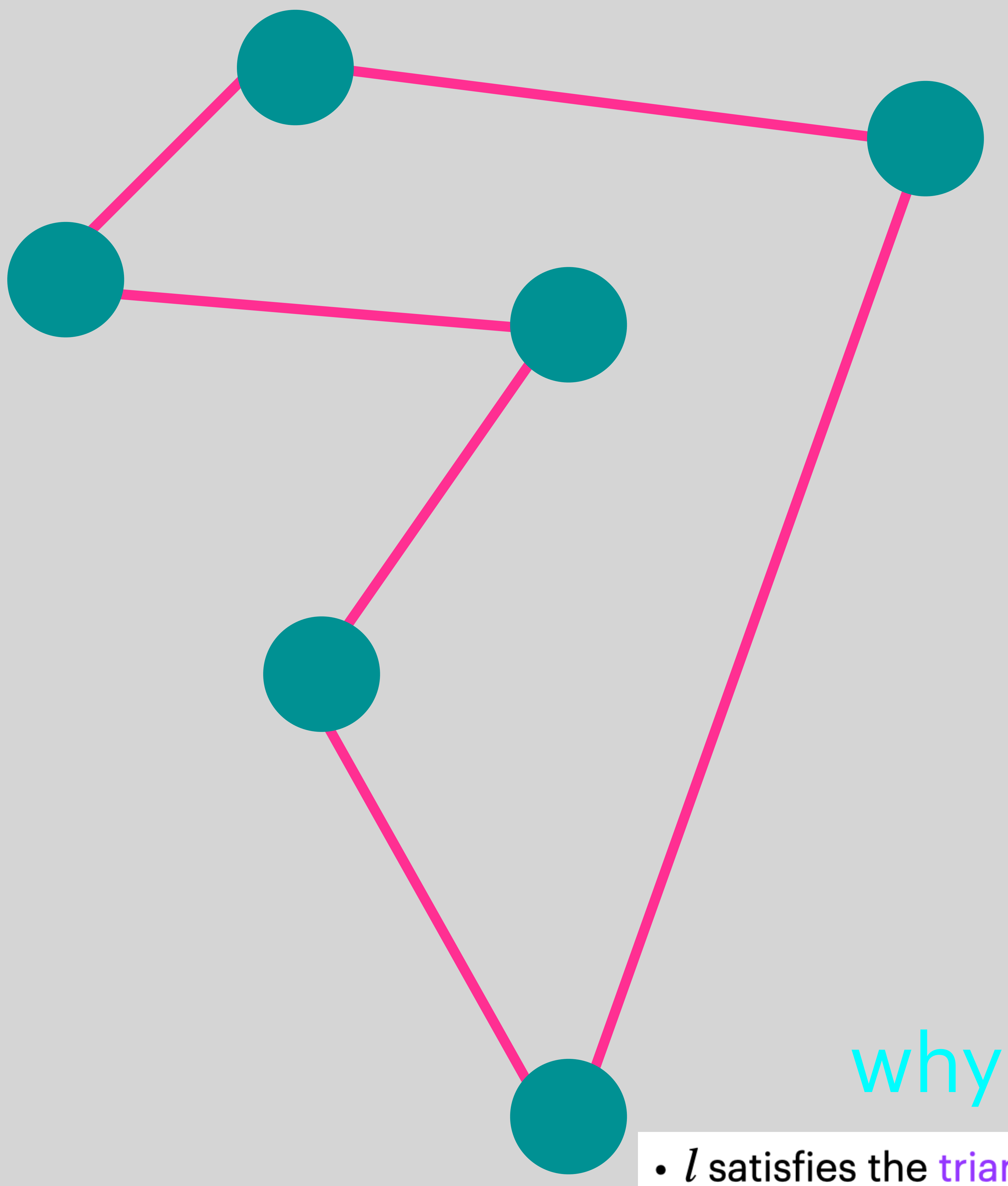
why ?

- l satisfies the triangle inequality
  $l(x,z) \leq l(x,y) + l(y,z)$

$$l(C) \leq l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once

⇒ Hamiltonian Cycle C

why ?

- l satisfies the triangle inequality

$l(x,z) \leq l(x,y) + l(y,z)$

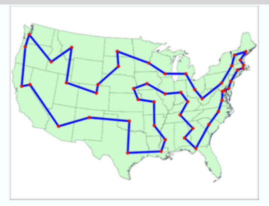
$$l(C) \leq l(W) = 2\,l(T) \leq 2\,OPT(K_n, l)$$

Metric TSP : 2-Approximation

Correctness

Goal :

Metric TSP : 2-Approximation
Problem Description

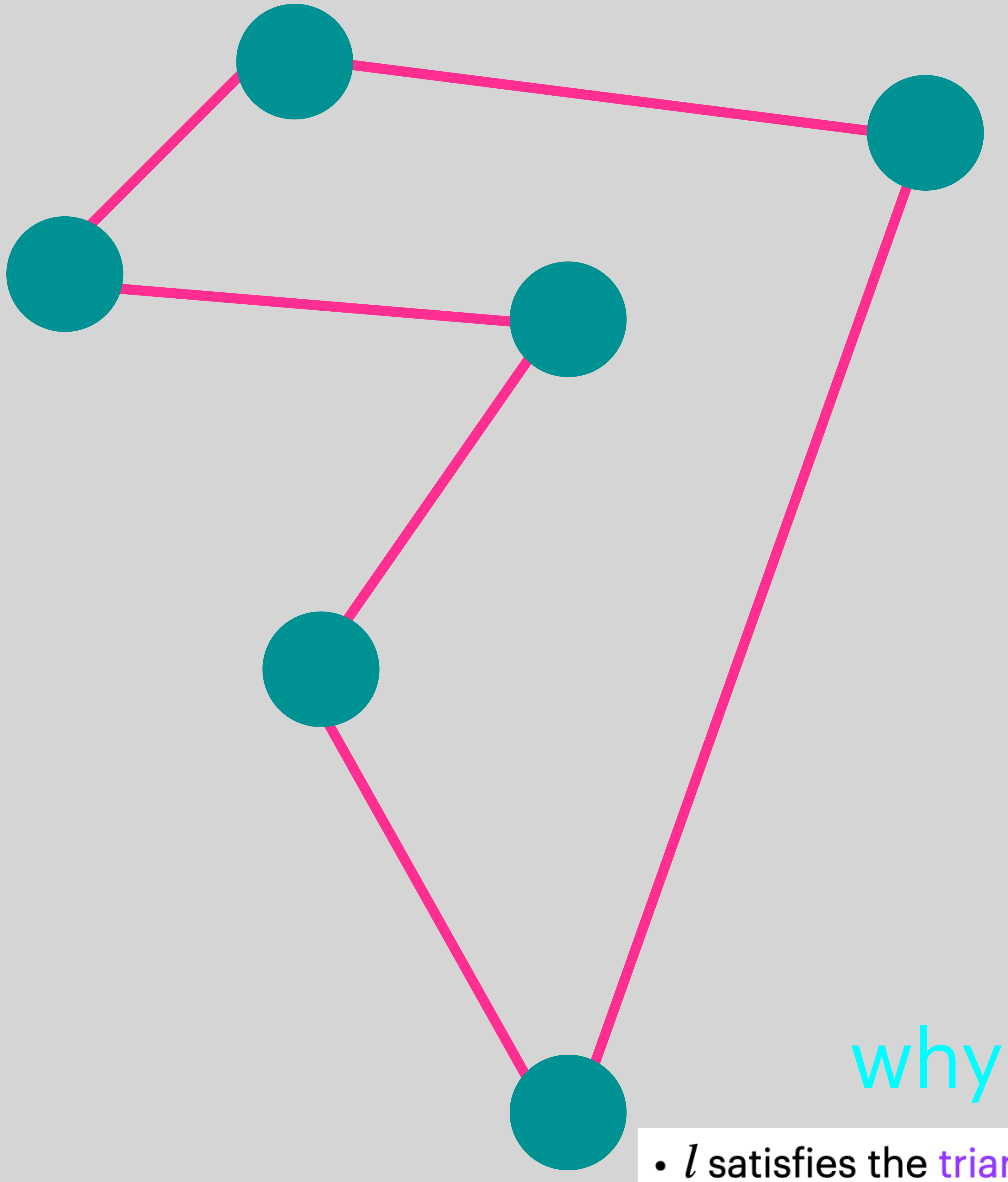


Given :
- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$
  - l satisfies the triangle inequality $l(x,z) \leq l(x,y) + l(y,z)$

To find :
- Hamiltonian Cycle C s.t.

$$l(C) \leq 2\, l(OPT)$$

where $OPT = \min_{H : \text{Hamiltonian Cycle}} \sum_{e \in E(H)} l(e)$



1. Find the MST T $\qquad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\qquad 2\, l(T) \leq 2\, OPT(K_n, l)$

3. Find Eulerian Tour W

$$l(W) = 2\, l(T) \leq 2\, OPT(K_n, l)$$

4. Traverse W once using shortcuts

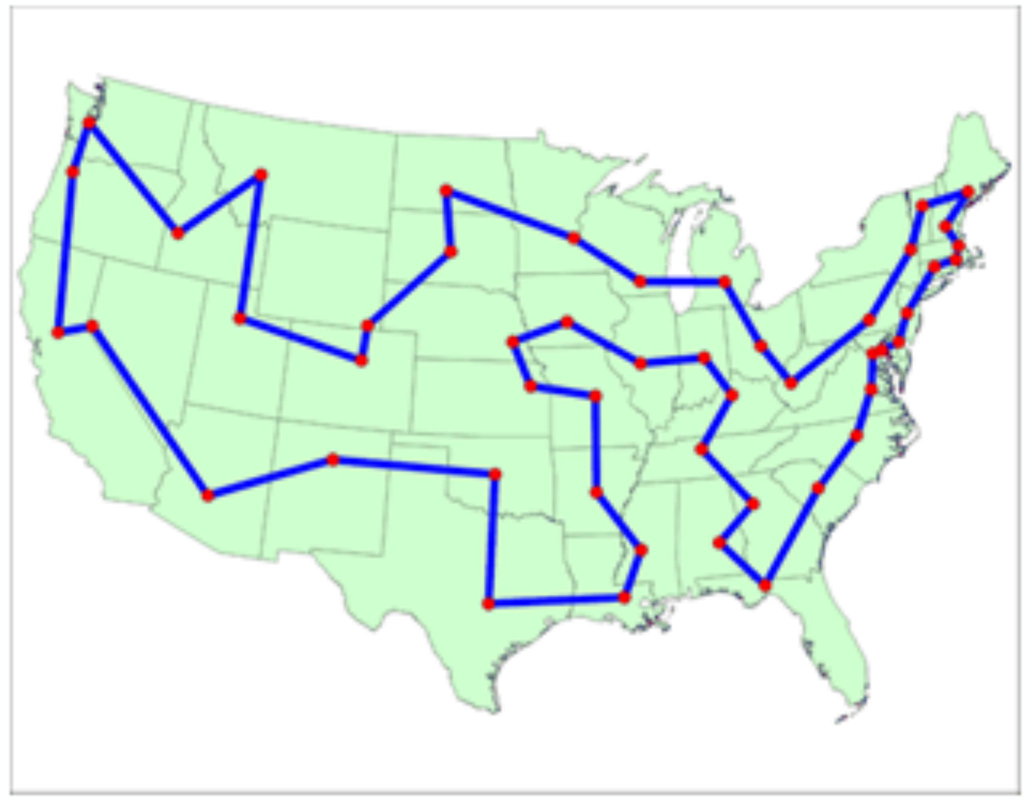
s.t. each vertex is visited exactly once $\Rightarrow$ Hamiltonian Cycle C

why ?

- l satisfies the triangle inequality $l(x,z) \leq l(x,y) + l(y,z)$

$$l(C) \leq l(W) = 2\, l(T) \leq 2\, OPT(K_n, l)$$

Metric TSP : 1.5-Approximation
Problem Description



Given :

- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$

- l satisfies the triangle inequality

$$l(x,z) \leq l(x,y) + l(y,z)$$

To find :

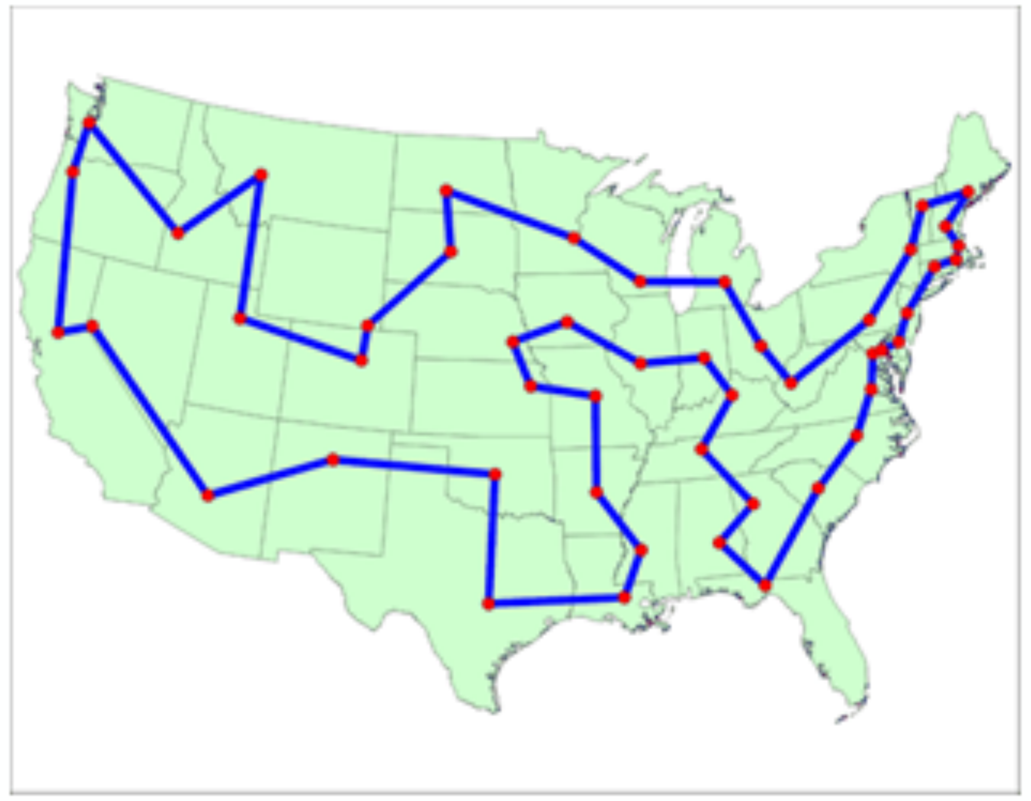
- Hamiltonian Cycle C s.t.

$$l(C) \leq 1.5\ l(OPT)$$

where $OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 1.5-Approximation

Problem Description



Given :

- A complete Graph K_n of n vertices
- Distances l inbetween every 2 vertex $l : \binom{[n]}{2} \to R$

To find :

- Hamiltonian Cycle C s.t.

$$l(C) \leq 1.5\ l(OPT)$$

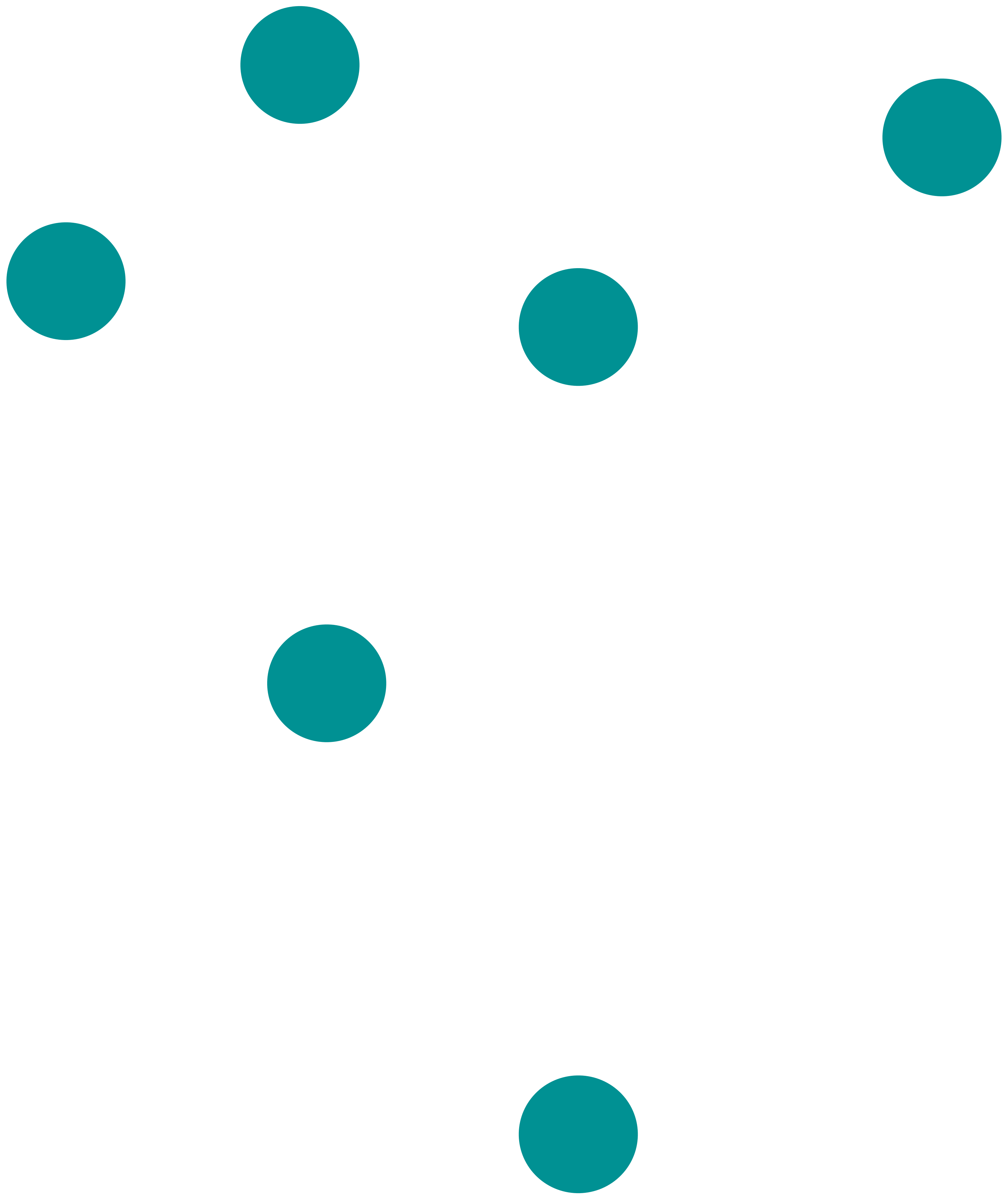
- l satisfies the triangle inequality

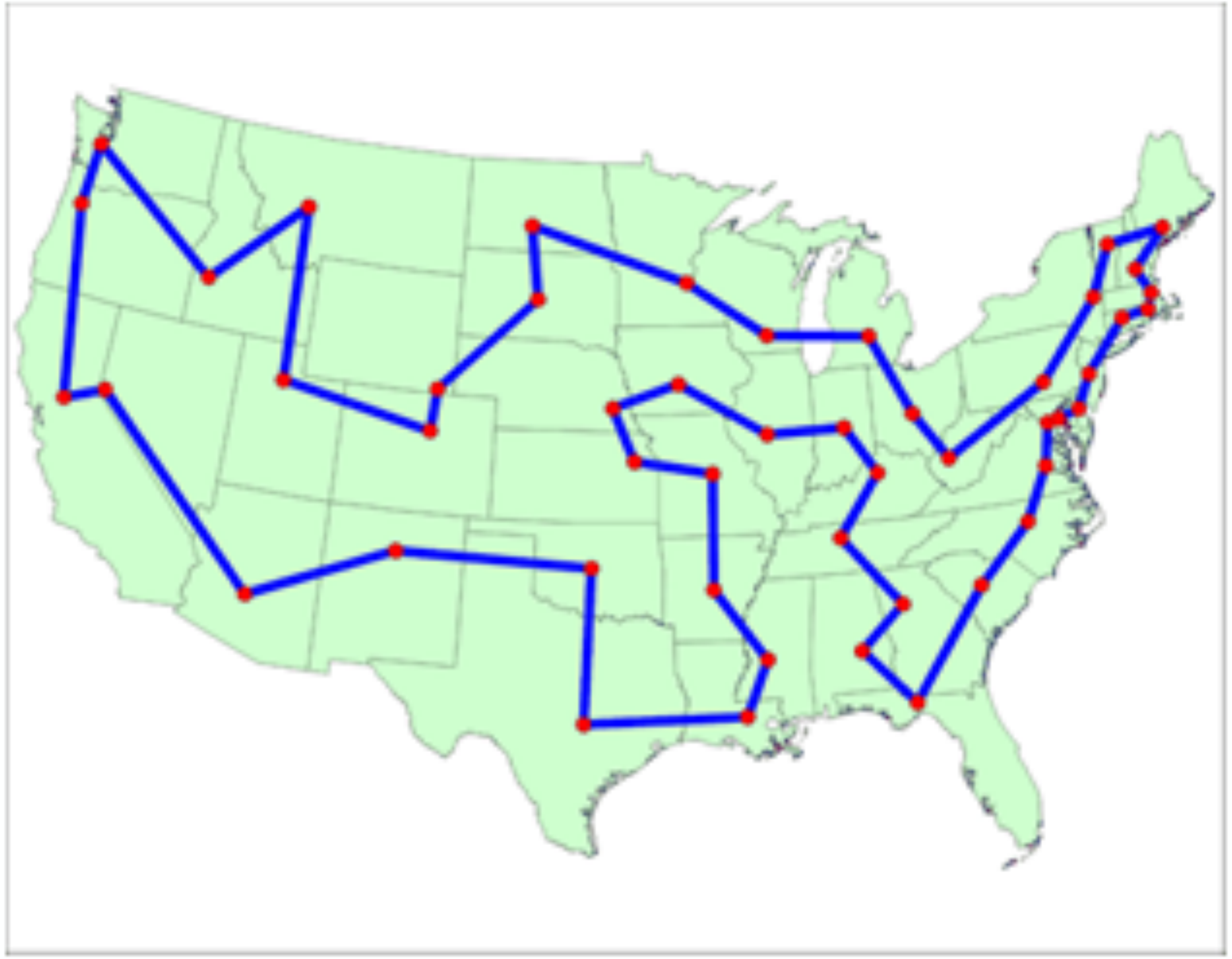
$$l(x,z) \leq l(x,y) + l(y,z)$$

where $OPT = \min_{H\,:\,Hamiltonian\ Cycle} \sum_{e \in E(H)} l(e)$

Metric TSP : 1.5-Approximation

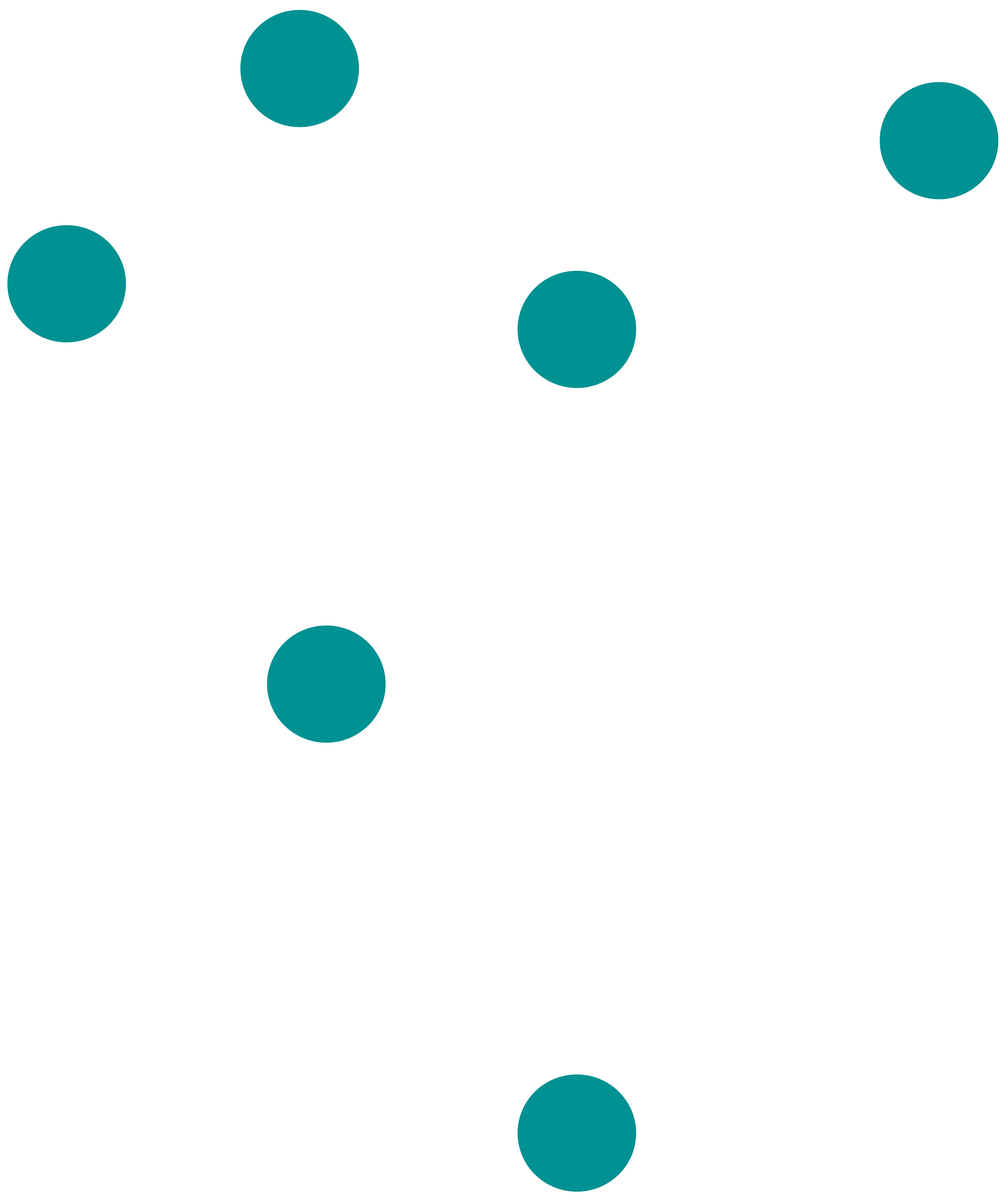
Algorithm





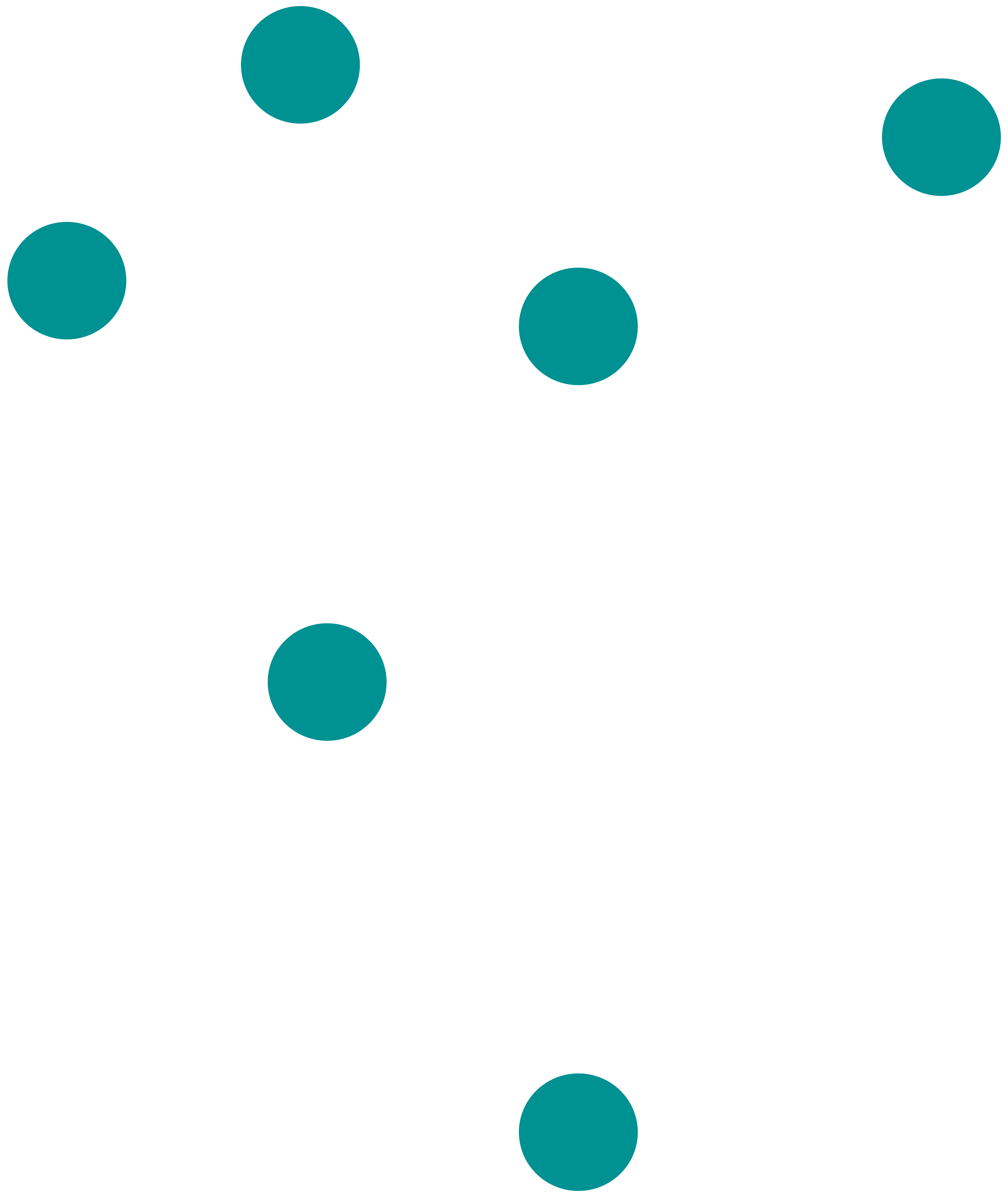
Metric TSP : 1.5-Approximation

Algorithm



Metric TSP : 1.5-Approximation

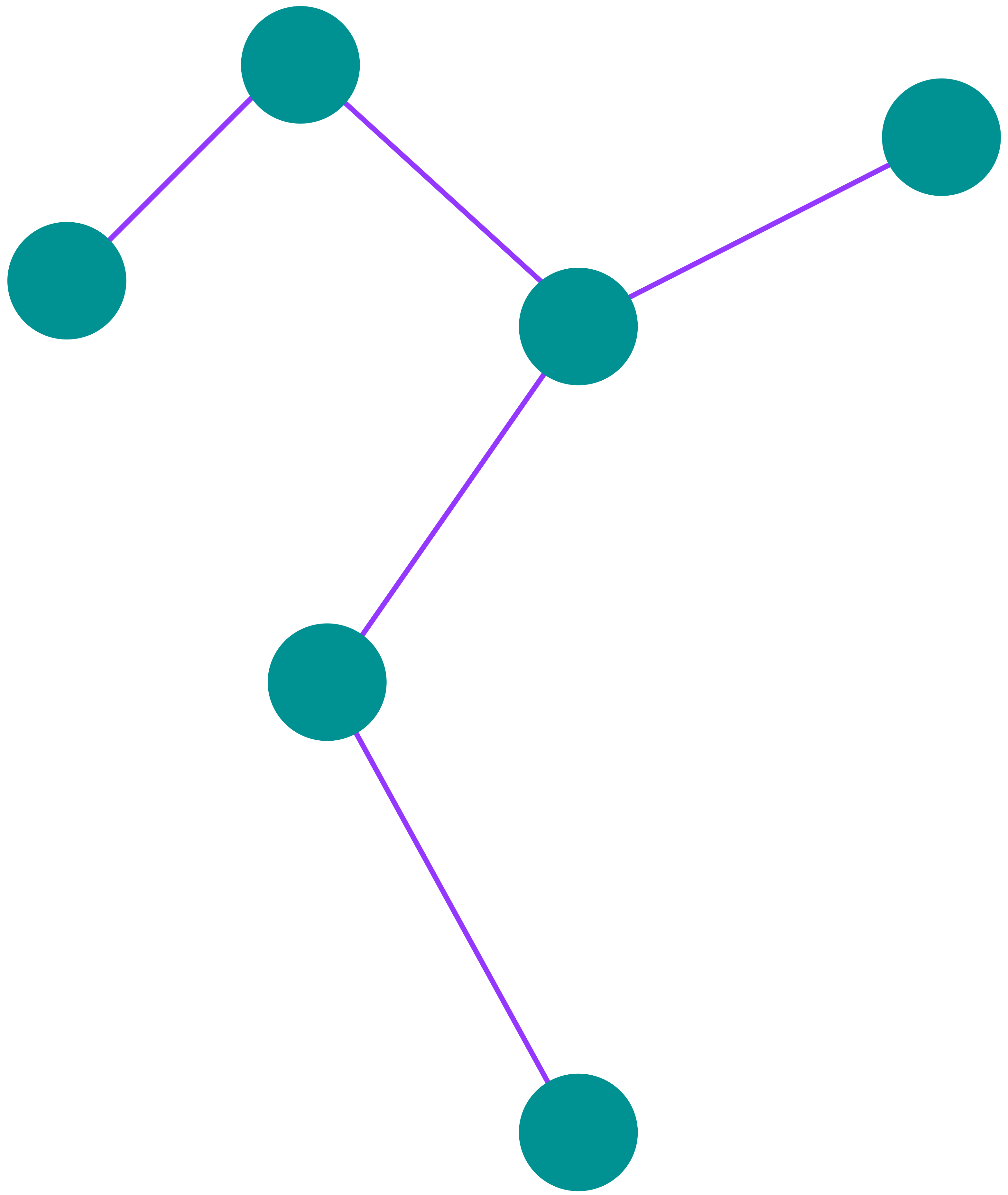
Algorithm



1. Find the MST T

Metric TSP : 1.5-Approximation

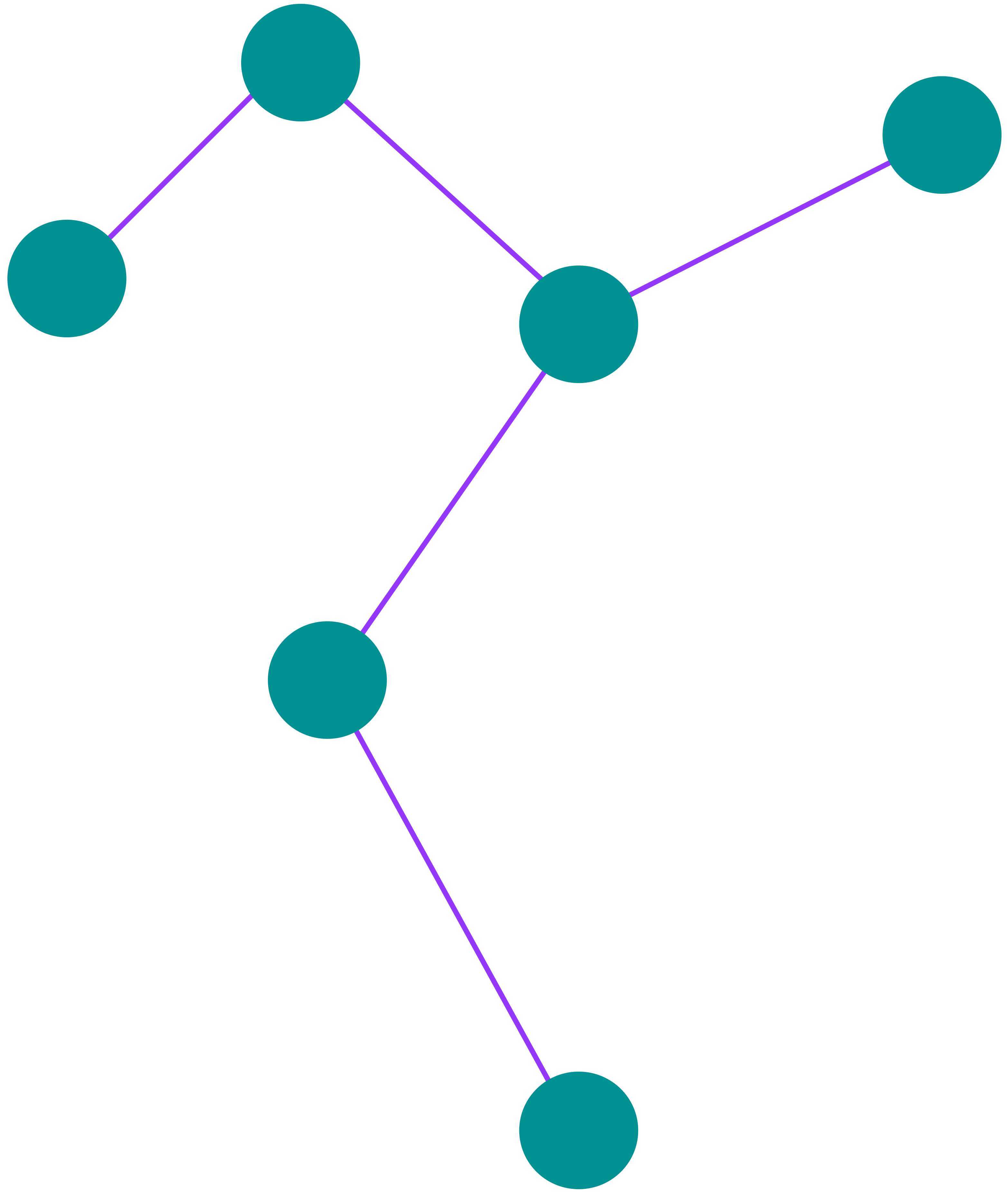
Algorithm



1. Find the MST T

Metric TSP : 1.5-Approximation

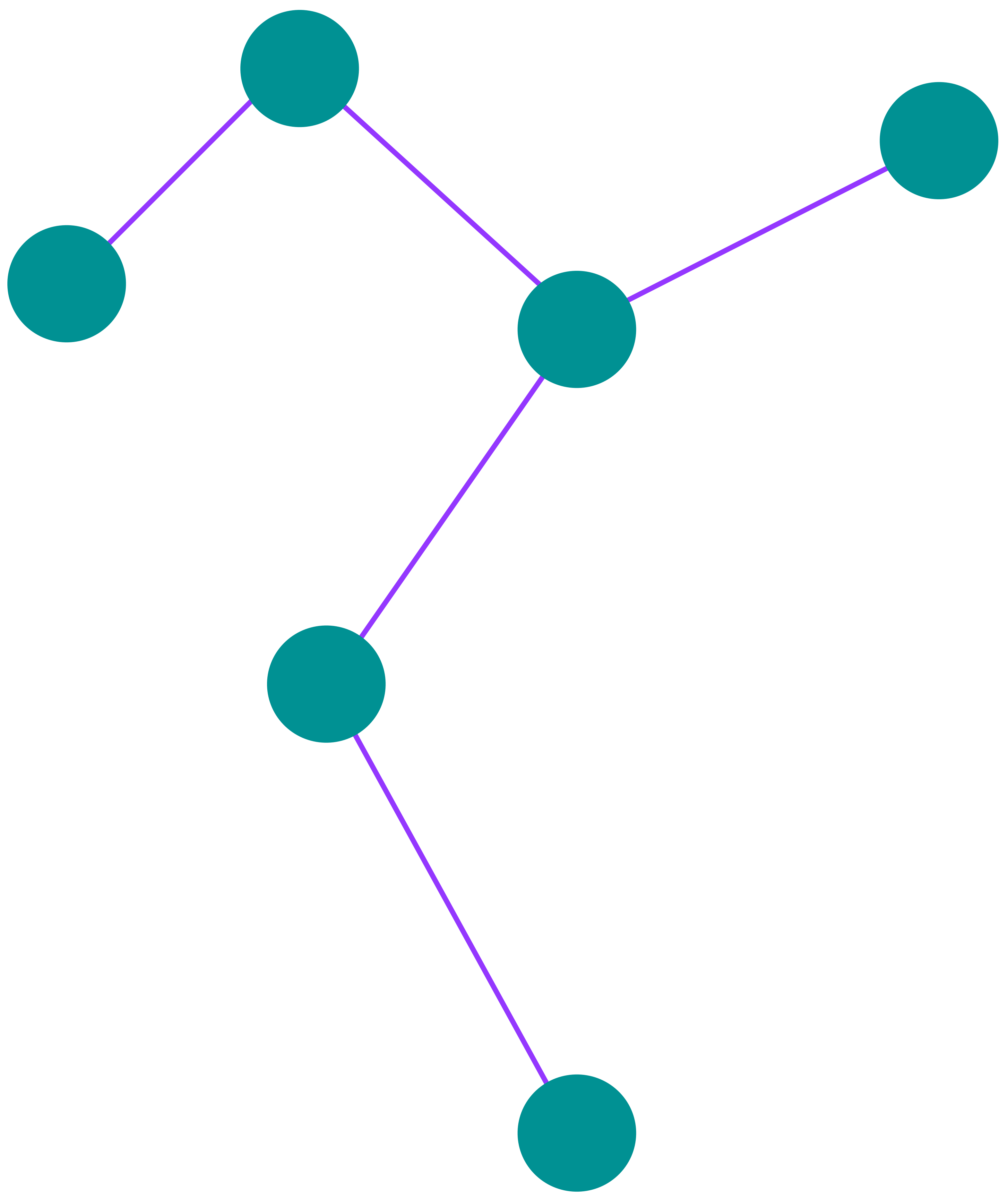
Algorithm



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

Metric TSP : 1.5-Approximation

Algorithm



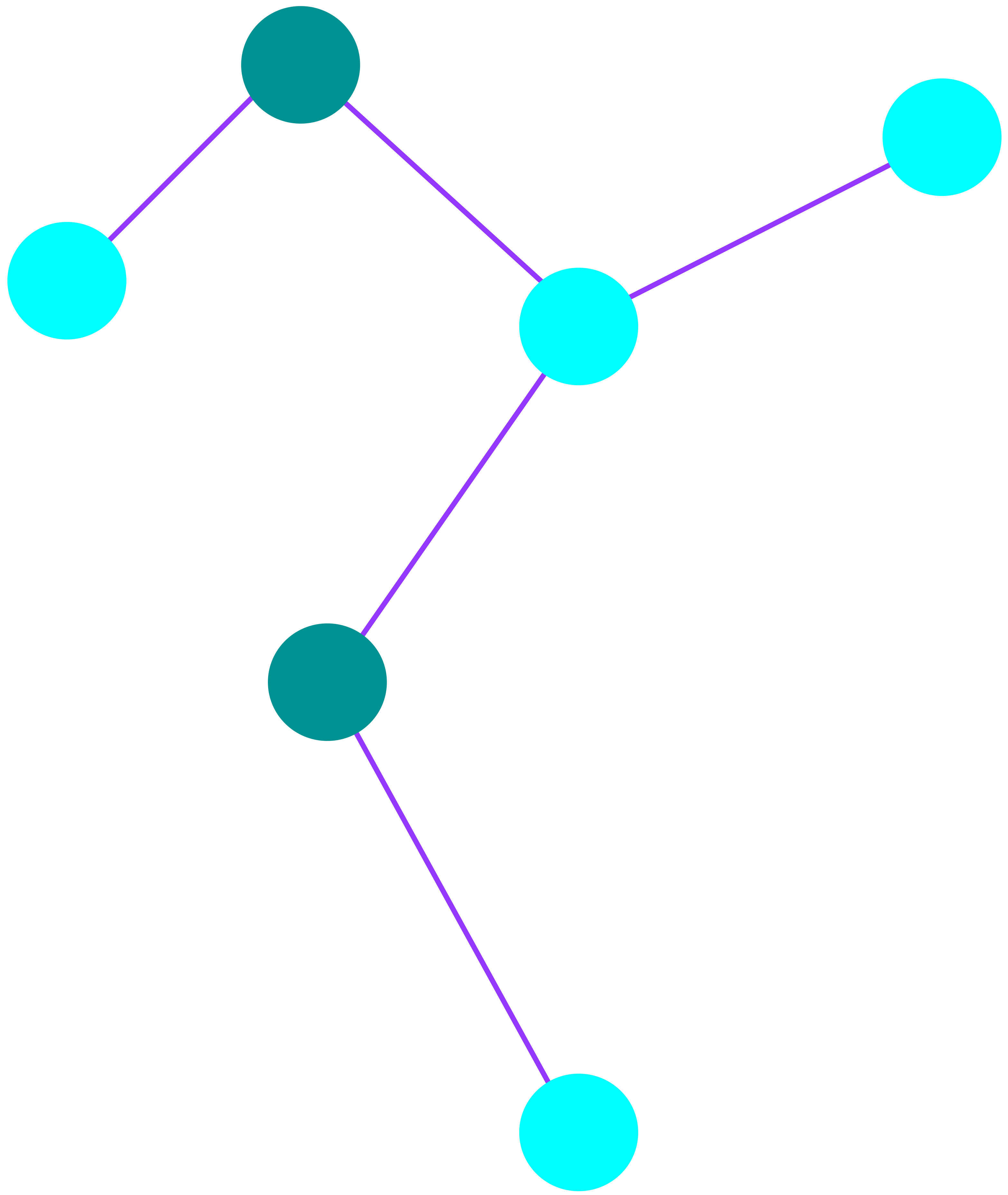
1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\, l(T) \leq 2\, OPT(K_n, l)$

2′. X:= Vertices with odd degree in T
Find minimal Matching M for X

Metric TSP : 1.5-Approximation

Algorithm



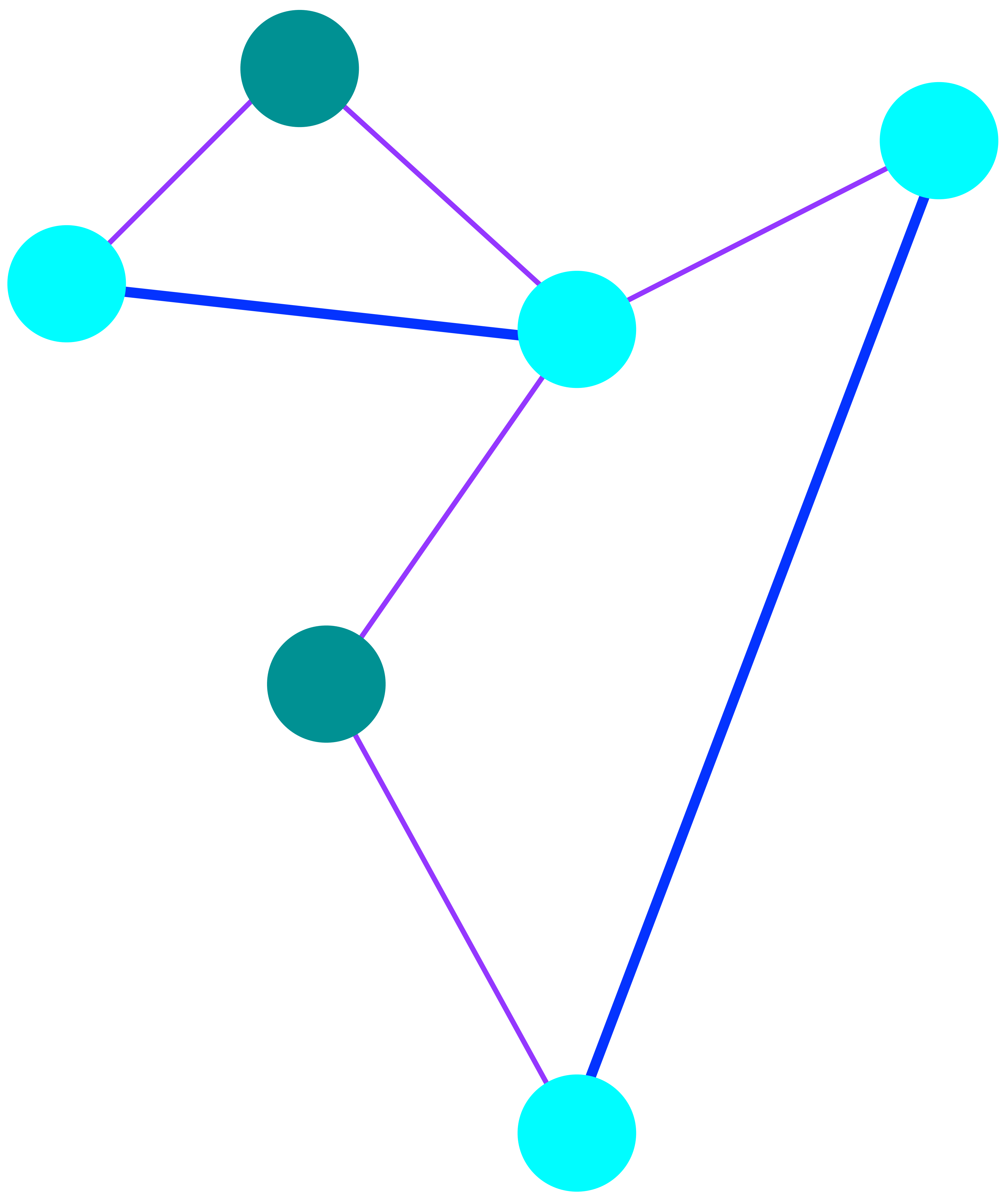
1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2. Duplicate all edges of T $\quad 2\, l(T) \leq 2\, OPT(K_n, l)$

2′. X:= Vertices with odd degree in T
Find minimal Matching M for X

Metric TSP : 1.5-Approximation

Algorithm



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

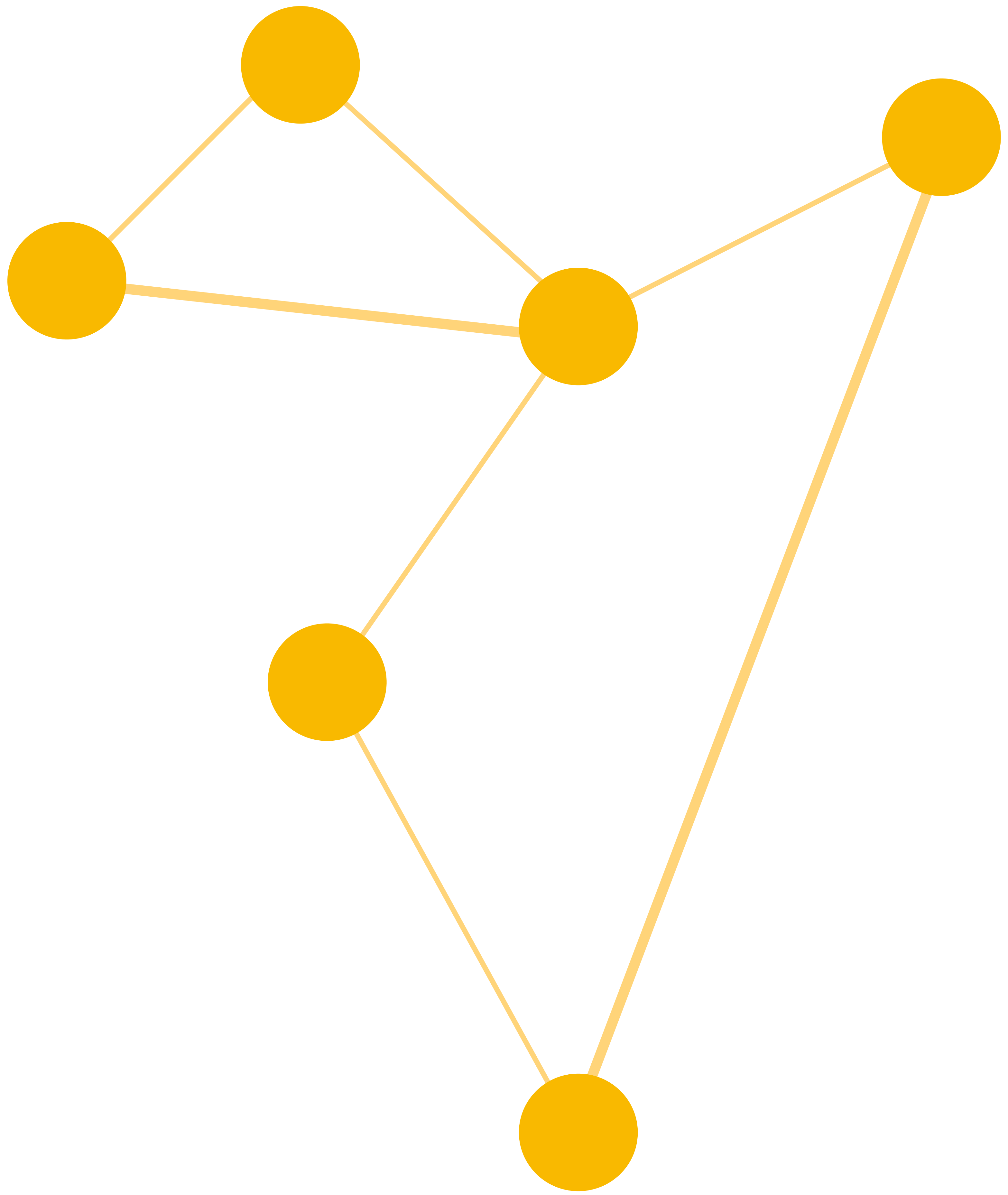
2. Duplicate all edges of T $\quad 2\,l(T) \leq 2\,OPT(K_n, l)$

2′. X:= Vertices with odd degree in T
Find minimal Matching M for X

$$l(M) \leq \frac{1}{2}\, OPT(K_n, l)$$

Metric TSP : 1.5-Approximation

Algorithm



1. Find the MST T $\quad l(T) \leq OPT(K_n, l)$

2′. X:= Vertices with odd degree in T

Find minimal Matching M for X

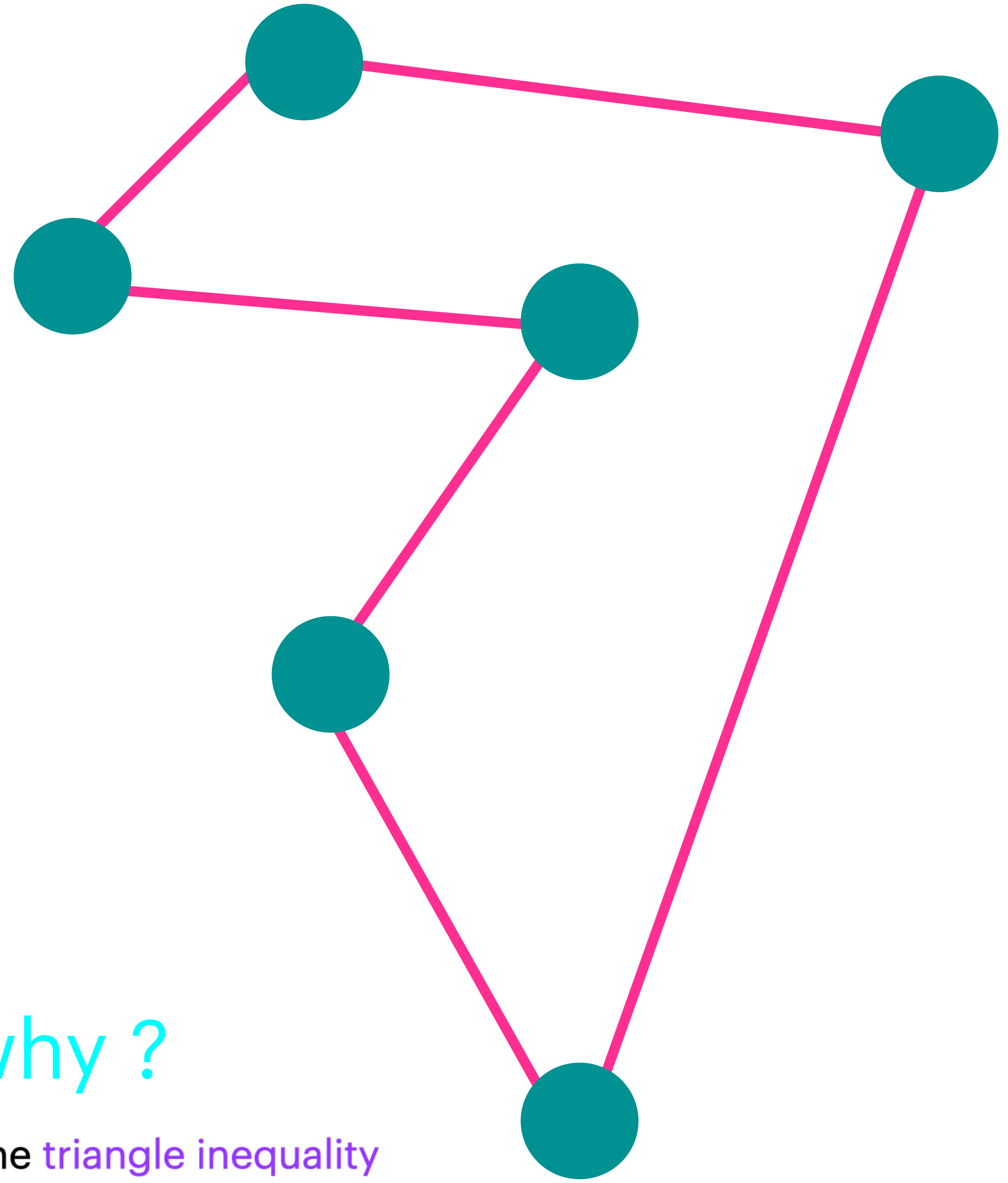
$$l(M) \leq \frac{1}{2} OPT(K_n, l)$$

3. Find Eulerian Tour W

$$l(W) = l(T) + l(M) \leq 1.5\, OPT(K_n, l)$$

Metric TSP : 1.5-Approximation

Algorithm



1. Find the MST T $\qquad l(T) \leq OPT(K_n, l)$

2′. X:= Vertices with odd degree in T

Find minimal Matching M for X

$$l(M) \leq \frac{1}{2}\ OPT(K_n, l)$$

3. Find Eulerian Tour W

$$l(W) = l(T) + l(M) \leq 1.5\, OPT(K_n, l)$$

4. Traverse W once using shortcuts

s.t. each vertex is visited exactly once

$\Rightarrow$ Hamiltonian Cycle C

why ?

- l satisfies the triangle inequality

$l(x,z) \leq l(x,y) + l(y,z)$

$$l(C) \leq l(W) = l(T) + l(M) \leq 1.5\, OPT(K_n, l)$$

Questions

Feedbacks , Recommendations

Nil Ozer