Introduction to Algorithms

Meng-Tsung Tsai

12/17/2019

Approximation Algorithms

Why do we need approximation algorithms?

Why do we need approximation algorithms?

In practice, we may not have sufficient computational resources to compute the exact (optimal) solution.

Computation resources could be:
(1) computation time
(2) memory space
(3) communication cost
...

Why do we need approximation algorithms?

Problem A is NP-hard, and thus it needs superpolynimal time unless P = NP.

If an approximate solution is good enough,
we may reduce the computation time.

Why do we need approximation algorithms?

Problem B (a two-party game) has communcation complexity $\Omega(n^2)$, and thus solving B in the streaming model requires memory space of $\Omega(n^2)$ bits.

If an approximate solution is good enough, we may reduce the amount of memory space.

Approximation ratio

We assume that the feasible solutions to our optimization problems are positive values. Let C* be the optimal solution, and let C be the found approximate solution.

For a maximization problem, the (multiplicative) approximation ratio is defined to be $(C^*/C) \geq 1$.

For a minimization problem, the (multiplicative) approximation ratio is defined to be $(C/C^*) \geq 1$.

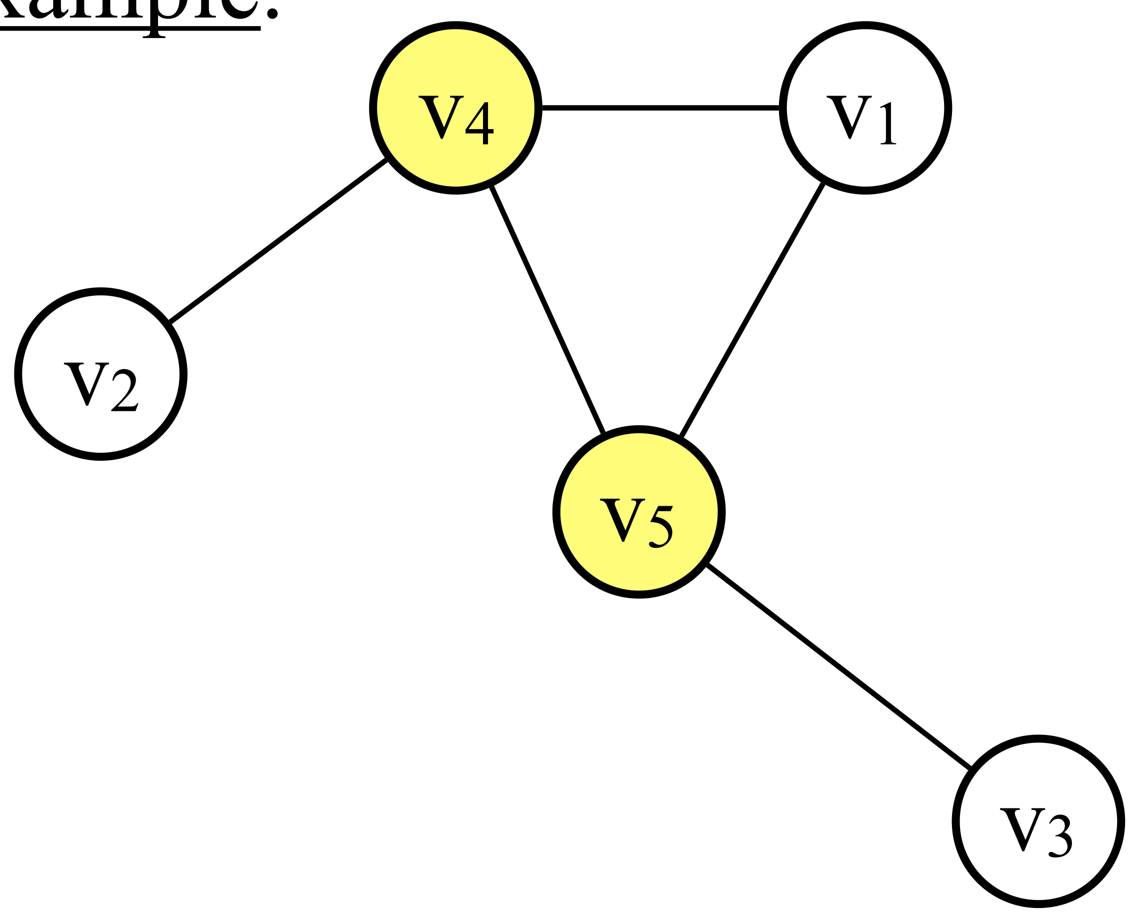
Examples

Minimum Vertex Cover

Input: an undirected graph G = (V, E).

Output: the size of a minimum-cardinality subset S of V so that for every edge (u, v) in E, node u or node v (maybe both) is in S.

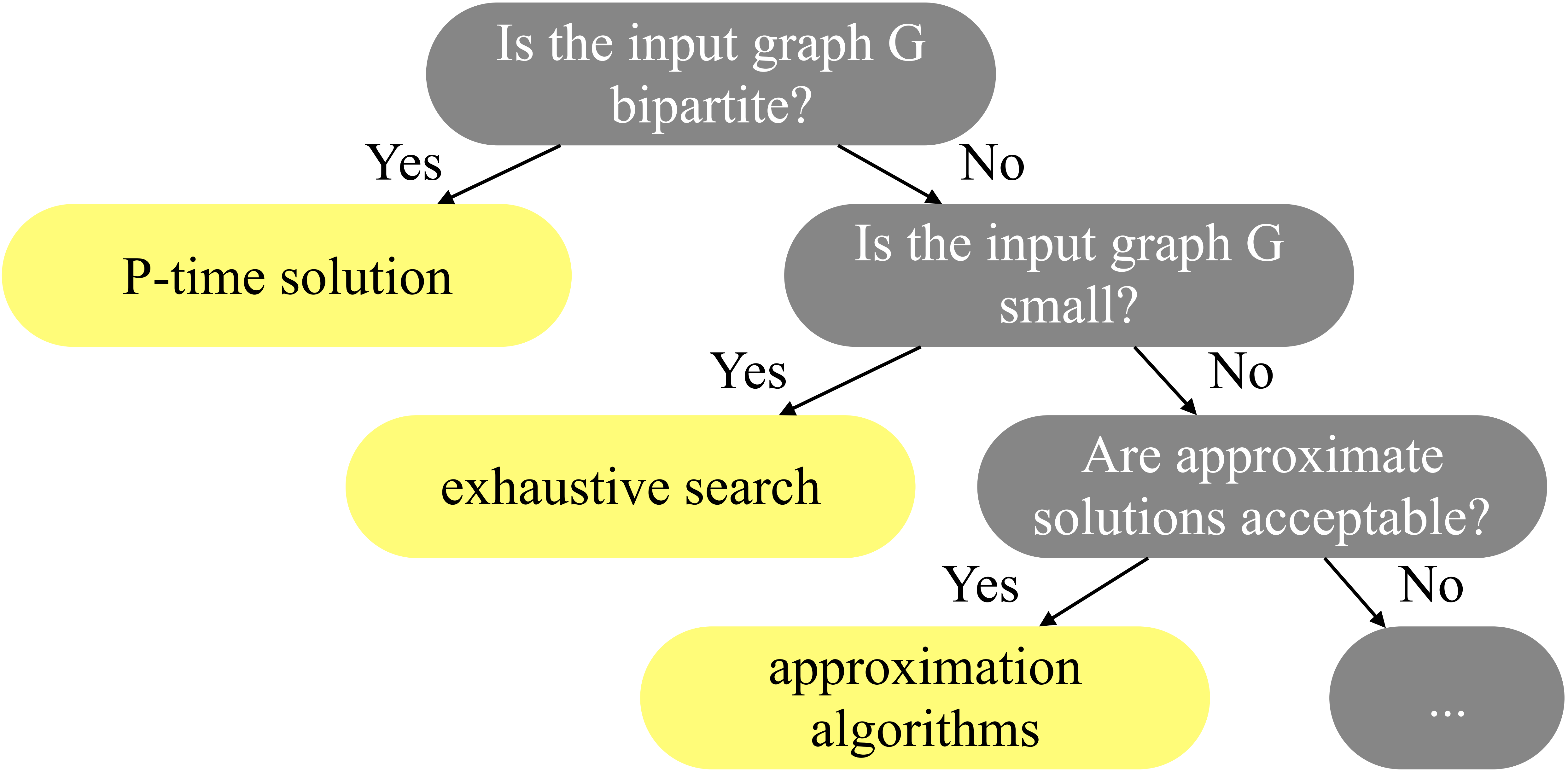
<u>Example</u>.



A vertex cover of cardinality (size) 2.

Minimum Vertex Cover

Recall that for general graphs Minimum Vertex Cover is NP-hard, and for bipartite graphs it has a P-time solution.



Minimum Vertex Cover

The following greedy algorithm has the approximation ratio 2. Alternatively, we could say, it is a 2-approximation algorithm.

```
S ← ∅, X ← ∅; // S is the set cover

while (there exists an edge eᵢ = (u, v) so that u ∉ S and v ∉ S){
    X ← X ∪ {eᵢ}; // a dummy operation
    S ← S ∪ {u} ∪ {v};
}

return |S|;
```

Minimum Vertex Cover

The following greedy algorithm has the approximation ratio 2.
Alternatively, we could say, it is a 2-approximation algorithm.

```
S ← ∅, X ← ∅; // S is the set cover

while (there exists an edge e_i = (u, v) so that u ∉ S and v ∉ S){
    X ← X ∪ {e_i}; // a dummy operation
    S ← S ∪ {u} ∪ {v};
}

return |S|;
```

Note that X is a matching.

Minimum Vertex Cover

The greedy algorithm has the approximation ratio 2.
Alternatively, we could say, it is a 2-approximation algorithm.

Proof.

Let S^* be the minimum vertex cover. For each edge (u, v) in X, u in S^* or v in S^*. Since X is a matching, no element in S^* can cover more than one edges in X. Therefore, $|S^*| \geq |X|$.

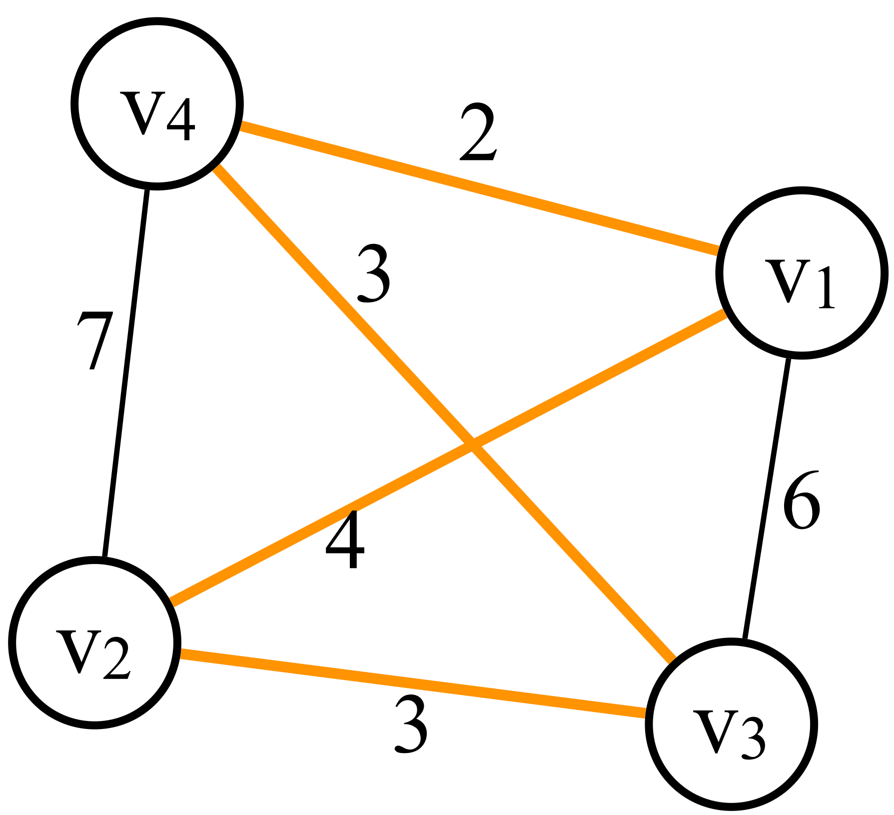
Together with $|S| = 2|X|$, we have $|S| \leq 2|S^*|$. Thus, the approximation ratio is at most 2.

Travelling Salesman Problem

Input: a complete undirected graph $G = (V, E)$, in which each edge e is associated with a non-negative weight $w(e) \geq 0$.

Output: a simple cycle C that traverses all nodes in G so that $\sum_{e \in C} w(e)$ is minimized.

Example.



A TS tour of weight 12.

Travelling Salesman Problem

Input: a complete undirected graph $G = (V, E)$, in which each edge e is associated with a non-negative weight $w(e) \geq 0$.

Output: a simple cycle C that traverses all nodes in G so that $\sum_{e \in C} w(e)$ is minimized.

We assume that the input graph satisfies the triangle inequality; i.e., for all $u, v, x \in V$, $w(u, v) + w(v, x) \geq w(u, x)$.

Travelling Salesman Problem

Input: a complete undirected graph $G = (V, E)$, in which each edge e is associated with a non-negative weight $w(e) \geq 0$.

Output: a simple cycle C that traverses all nodes in G so that $\sum_{e \in C} w(e)$ is minimized.

We assume that the input graph satisfies the triangle inequality; i.e., for all $u, v, x \in V$, $w(u, v) + w(v, x) \geq w(u, x)$.

This assumption is satisfied for many applications on real maps.

Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow \mathrm{MST}(G)$; // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

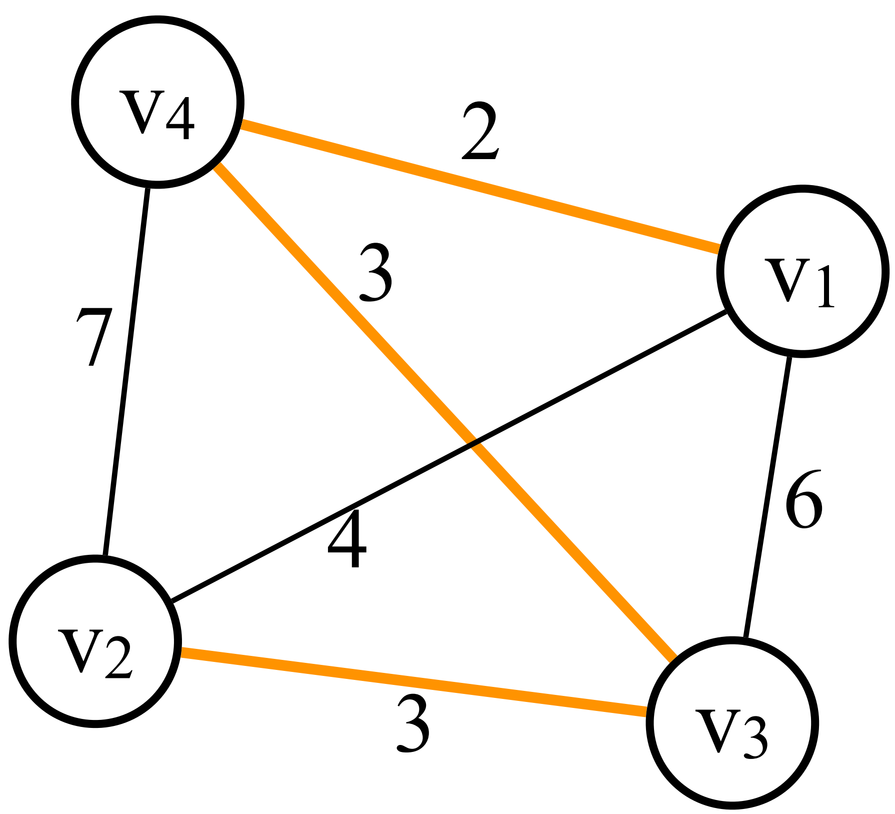
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow$ MST(G); // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

Example.



MST(G)

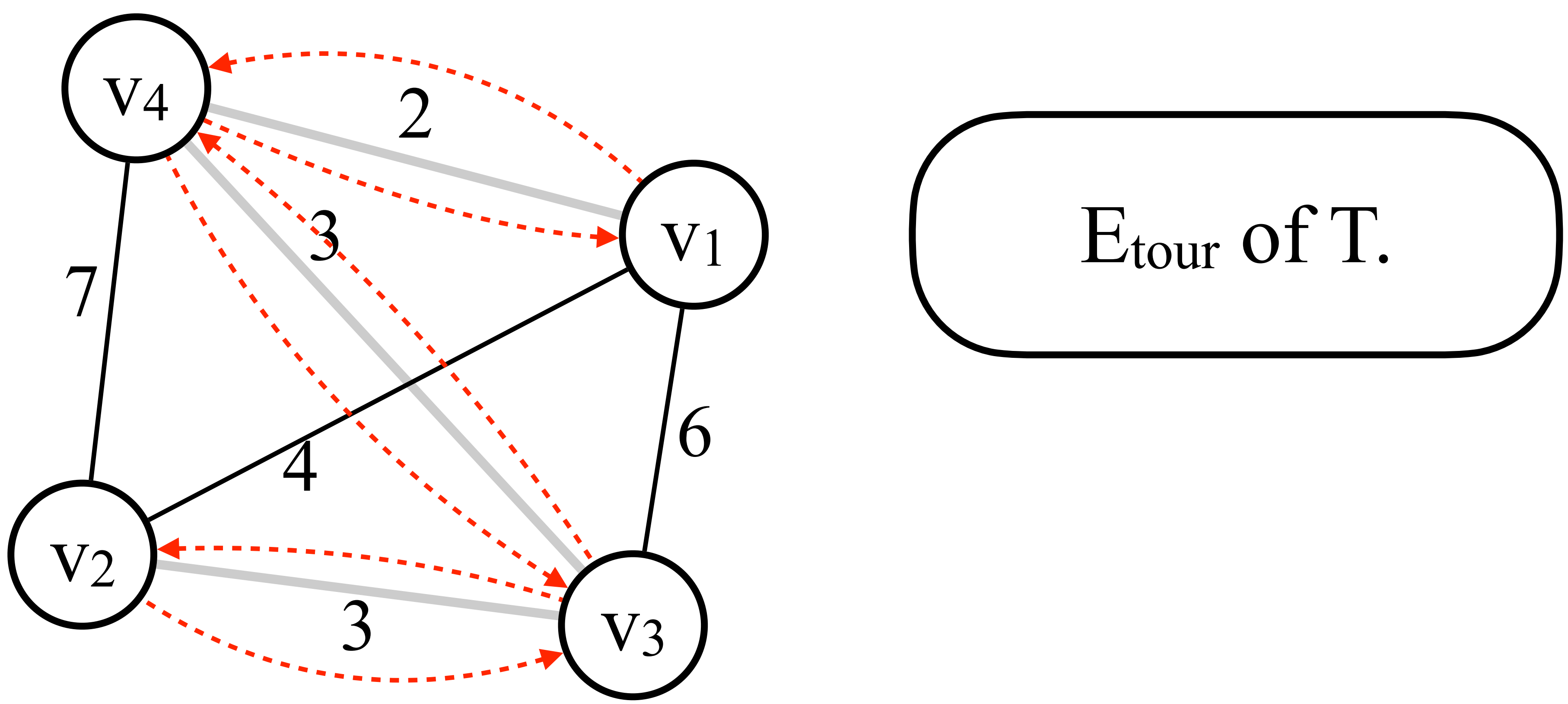
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow$ MST(G); // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

<u>Example</u>.



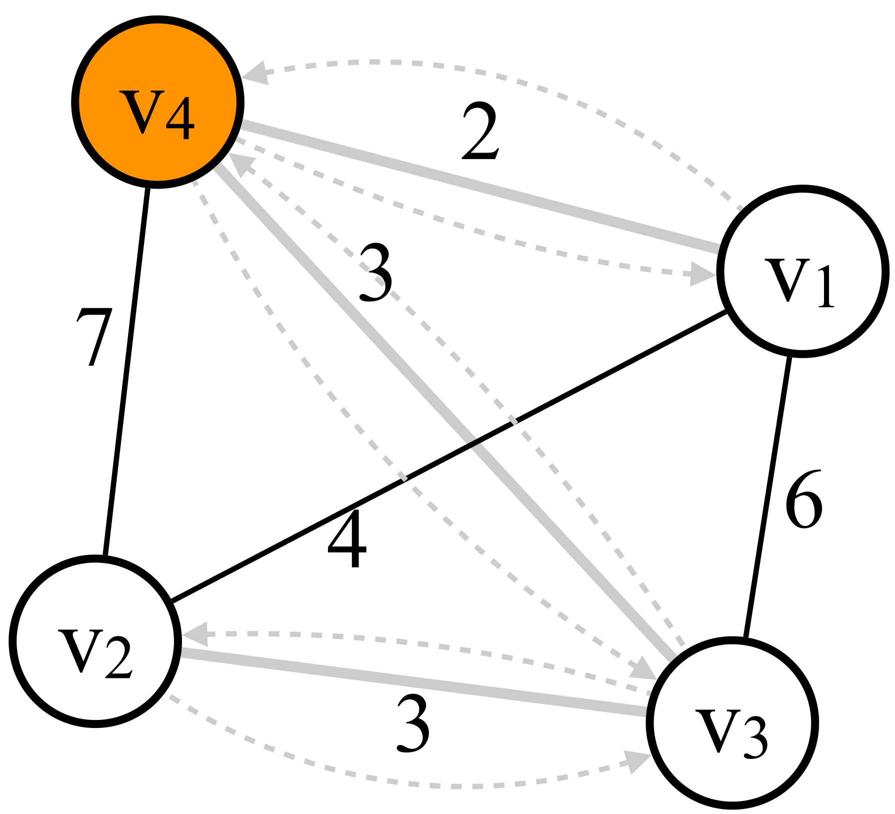
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow \text{MST}(G)$; // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

<u>Example</u>.



TS_{tour}.

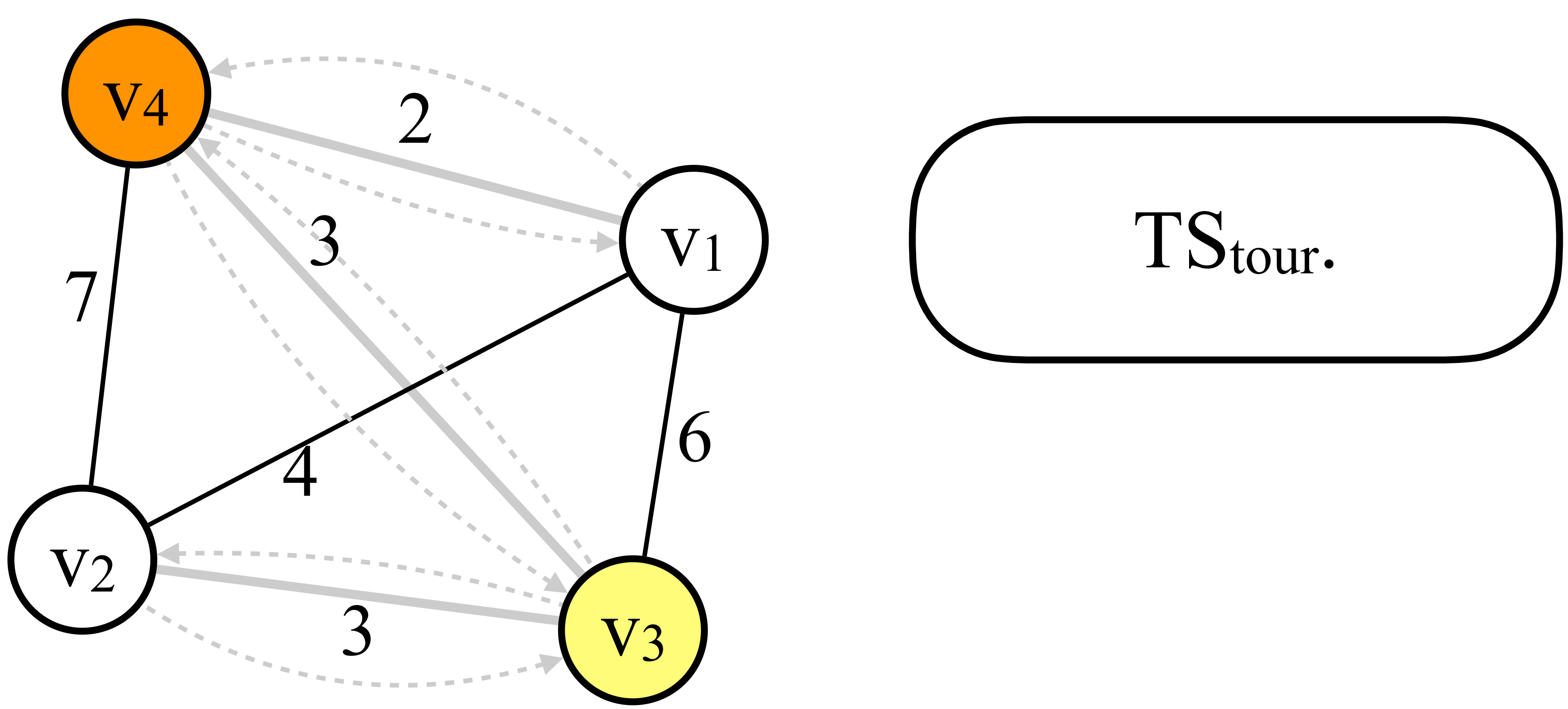
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow$ MST(G); // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

Example.



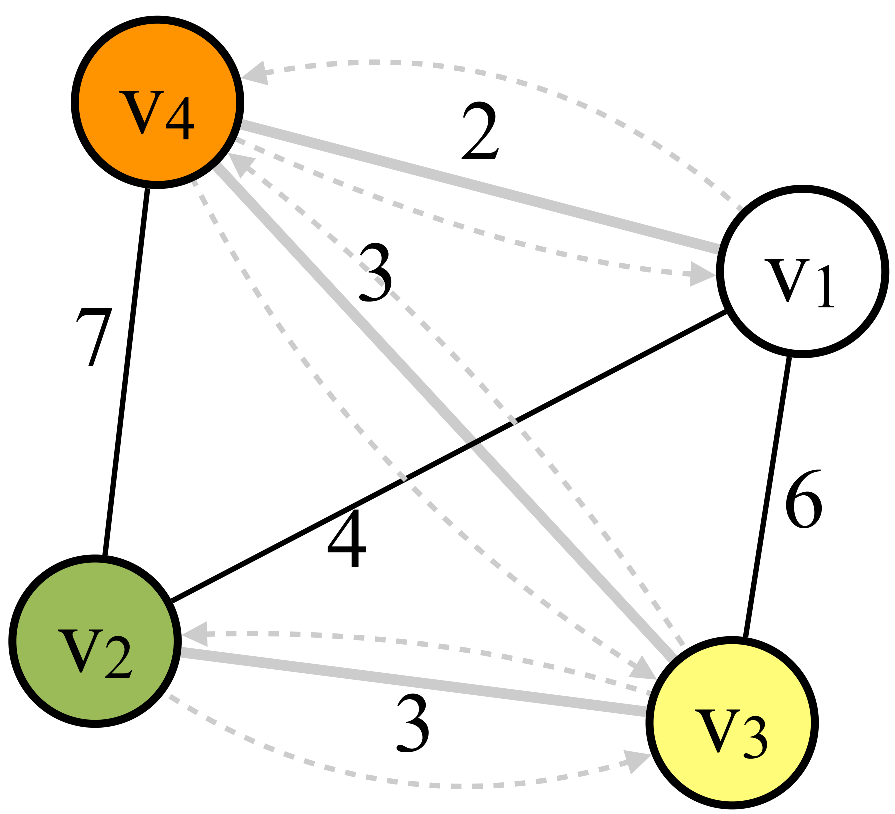
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow$ MST(G); // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

Example.



TS_{tour}.

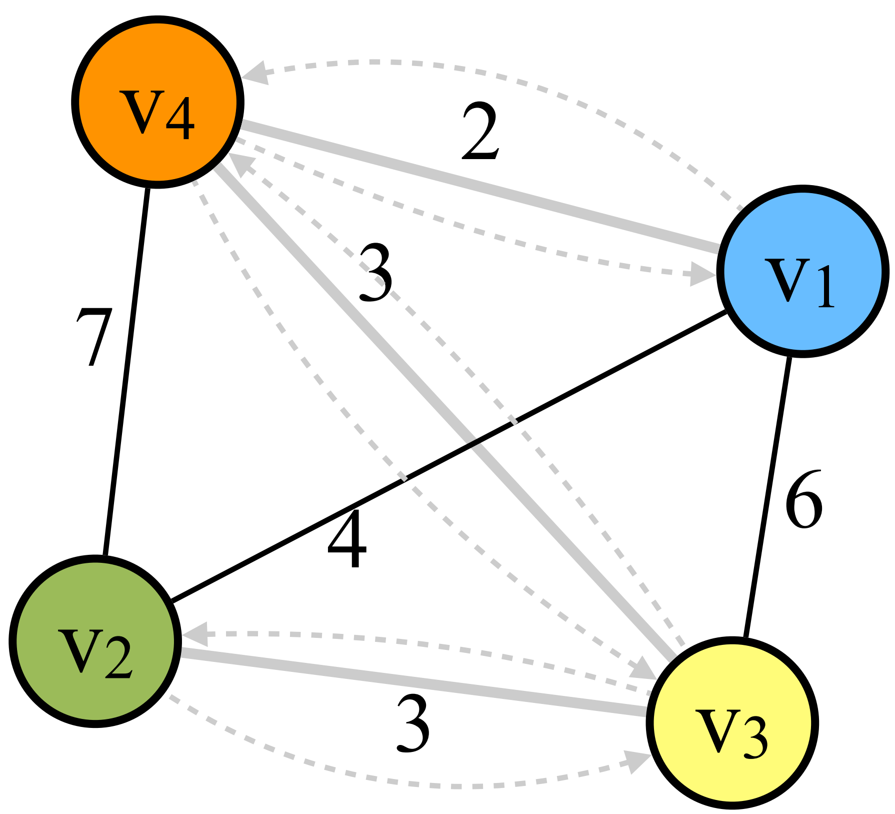
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow \text{MST}(G)$; // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

Example.



TS_{tour}.

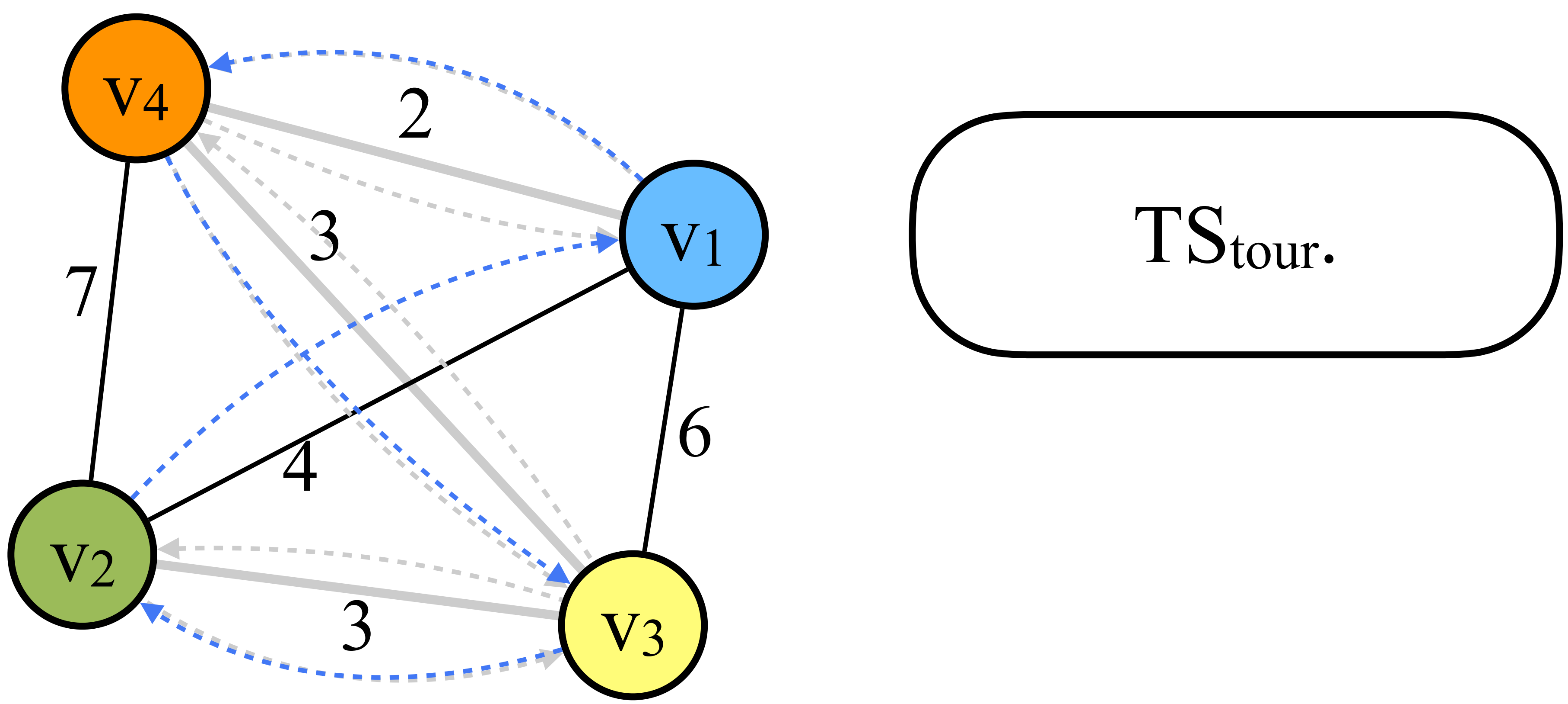
Travelling Salesman Problem

The following algorithm is a 2-approximation algorithm.

$T \leftarrow$ MST(G); // minimum spanning tree
$E_{tour} \leftarrow$ Eulerian tour of T;
$TS_{tour} \leftarrow$ remove repeated nodes from E_{tour};

return TS_{tour};

Example.



Travelling Salesman Problem

The algorithm is a 2-approximation algorithm.

Proof.

Let TS* be the minimum-weight travelling salesman tour. TS* can be seen as a spanning tree + an edge. Hence,

$$w(TS^*) \geq w(T)$$

because T is the minimum spanning tree and all $w(e) \geq 0$.

Travelling Salesman Problem

The algorithm is a 2-approximation algorithm.

Proof.

Let TS* be the minimum-weight travelling salesman tour. TS* can be seen as a spanning tree + an edge. Hence,

$$w(TS^*) \geq w(T)$$

because T is the minimum spanning tree and all $w(e) \geq 0$.

By the trianlge inequality, shortcutting the tour cannot increase the weight and thus

$$2w(T) = w(E_{tour}) \geq w(TS_{tour}).$$

Since TS* is the optimal TS-tour, $w(TS_{tour}) \geq w(TS^*)$. As a result, $w(TS_{tour}) \in [w(T), 2w(T)]$ and $w(TS_{tour}) \in [w(TS^*), 2w(TS^*)]$.

Maximum Matching

Input: an undirected graph G = (V, E)

Output: the size of a maximum-cardinality subset F of E so that no two edges in F share an endpoint.

This problem can be solved in $O(n^{1/2} m)$ time (i.e. has a P-time solution). If the running time is not affordable, one can obtain a 2-approximation of this problem in $O(n+m)$ time.

Maximum Matching

<u>Claim</u>. Let M* be a maximum matching, and let M be any maximal matching. Then, $|M| \geq |M^*|/2$.

Proof.

Let (u, v) be an edge in M*. Because M is a maximal matching, at least one of nodes u, v is an endpoing of some edge in M. Otherwise, M shall include edge (u, v). This implies that $|M| \geq |M^*|/2$.

Maximum Matching

The following greedy algorithm can return a maximal matching.

```
M ← ∅, X ← ∅;
Maximal-matching(G){
    foreach(edge (u, v) in G){
        if(u ∉ X and v ∉ X){
            X ← X ∪ {u} ∪ {v};
            M ← M ∪ {(u, v)};
        }
    }
}
return |M|;
```

The above procedure runs in $O(|V|+|E|)$ time.

Exercise: 2/3-approximation of Matching

```
Matching(G){

  M ← ∅;
  foreach(e){
      if(M ∪ {e} is a matching){
          M ← M ∪ {e};
      }
  }
  while(there exists a length-3 augmenting path P w.r.t. M){
      M ← M ⊕ P;
  }

  return M;
}
```

Exercise: 2/3-approximation of Matching

```
Matching(G){

  M ← ∅;
  foreach(e){
      if(M ∪ {e} is a matching){
          M ← M ∪ {e};
      }
  }
  while(there exists a length-3 augmenting path P w.r.t. M){
      M ← M ⊕ P;
  }

  return M;
}
```

$|M| \geq (2/3)\mathrm{OPT}$. (Why?)