# Module 6
# NP-Complete Problems and Heuristics

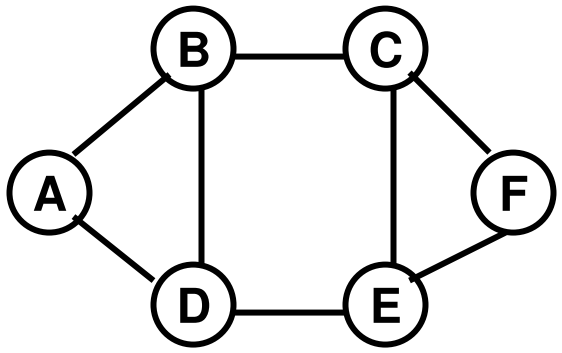
Dr. Natarajan Meghanathan
Professor of Computer Science
Jackson State University
Jackson, MS 39217
E-mail: natarajan.meghanathan@jsums.edu

## Optimization vs. Decision Problems

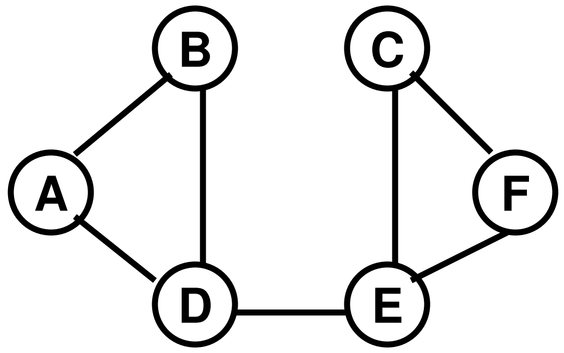
- A problem is an optimization problem if we want to maximize or minimize the solution.
  - Example: Minimum Spanning tree problem: Given a graph, we want to determine a spanning tree whose sum of the edge weights is the minimum.
- A problem is a decision problem if we want to get an Yes or NO answer as the solution.
  - Example: 2-Colorable (Bipartite) problem: Given a graph and two colors (say Black and White), we want to determine whether we can color the end vertices of each edge with the two different colors.

## Hamiltonian Circuit (HC) Problem

- Tour: A sequence of vertices such that the starting and ending vertex is the same, and the rest of the vertices appear exactly once in the sequence.



**Tour: A – B – C – F – E – D – A**

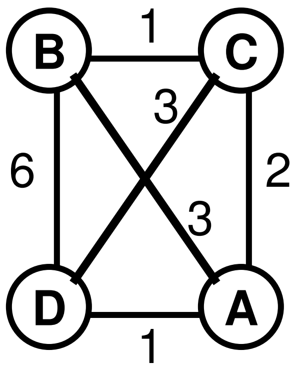


**There is NO tour for this graph**

- Given a graph (no edge weights), the Hamiltonian Circuit (HC) problem is to determine whether or not the graph has a tour.
- The HC problem is a decision problem.

# Traveling Salesman Problem (TSP)

- Given a graph with edge weights, the TSP problem is to determine a minimum weight tour (if a tour exists) for the graph.



| **Tours** | **Weight** |
|---|---|
| **A – B – C – D – A** | **8** |
| A – B – D – C – A | 14 |
| A – C – B – D – A | 10 |
| A – C – D – B – A | 14 |
| **A – D – C – B – A** | **8** |
| A – D – B – C – A | 10 |

**Decision version: Is there a tour of weight W?**

**Optimization version: Find a tour of the minimum possible weight?**

# P, NP Problems

- A problem is said to belong to the class NP (non-deterministic polynomial) if:
  - Step 1: We can determine (or at least guess) a solution for the optimization or decision version of a problem in polynomial time
  - Step 2: Check the correctness of the solution of Step 1 in polynomial time
- A problem in the class NP is said to also belong to the class P if (Step 1 is deterministic) the optimal/decision solution can be determined in polynomial time for all the input instances.
- That is, all problems of class P are also said to belong to the class NP, but not vice-versa (at least for now!!).
  - i.e., There exists several problems for which Step 1 is not deterministic. We cannot come up with an algorithm for such a problem that is guaranteed to give the optimal solution or decision solution for all the input instances.
- Example of problems in class P: Sorting, Shortest Path problem, Minimum Spanning tree problem, etc.
- Example of problems in class NP that do not belong to class P: Hamiltonian circuit problem, Traveling salesman problem (decision version), Maximum clique problem, etc.

## Decision Version of TSP is in NP

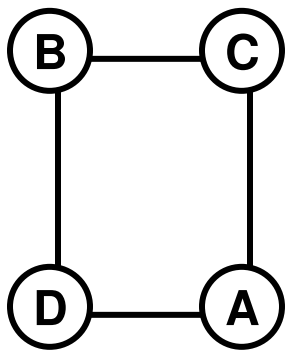
- The decision version of the TSP problem is in NP and not in P, because:
  - There is no polynomial-time algorithm for the decision version of the TSP problem that is guaranteed to say that a graph has a tour of a target weight (or a weight less than or equal to a target weight) even if the graph has such a tour.
  - Depending on how the algorithm is designed, it may give an YES answer for a graph and at least one such tour really exist in the graph; but for some other graph may give a NO answer even if such a tour exists in the graph.
  - Hence, we say all algorithms that currently exist for the decision version of the TSP problem are non-deterministic in nature (i.e., in class NP).

# Polynomial-Time Reduction: HC $\leq_P$ TSP

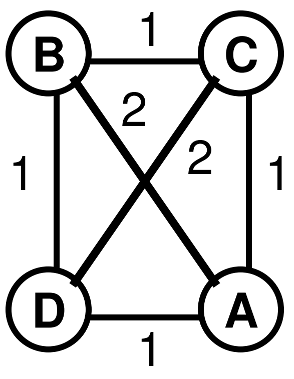
- HC Problem: Given a graph G = (V, E) with no edge weights, determine whether there exists a HC in G.
- Assumption: Let there be a deterministic polynomial-time algorithm for the decision version of the TSP problem.
- Reduction Step: Transform G = (V, E) to a weighted graph G* = (V, E*), where E* = V(V-1)/2, as follows:
  - For every pair (u, v) for which there exists an edge in G, create an edge (u, v) of weight 1 in G*
  - For every pair (u, v) for which there is NO edge in G, create an edge (u, v) of weight 2 in G*

  $\}$ **Time Complexity V(V-1)/2 = $\Theta(V^2)$**
- If G has a HC, then we should be able to find a minimum weight tour of total weight equal to the number of vertices (V) in G*.
  - The weight of each edge in the minimum weight tour will be 1.
- If G has no HC, then the minimum weight tour in G* would be of weight greater than V.
  - There should be at least an edge of weight 2 in the tour.
- Thus, we could find a polynomial-time solution for the HC problem if there exists a polynomial-time algorithm for the decision version of the TSP problem.

# HC $\leq_P$ TSP: Example

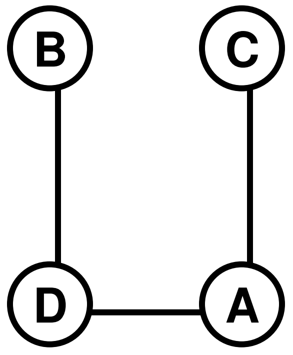


Graph G

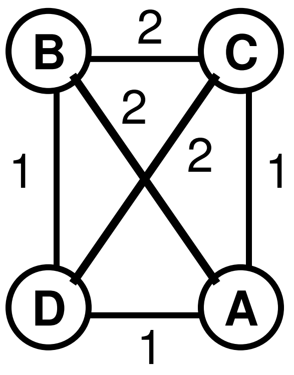


Graph G*

A – C – B – D – A
The weight of each edge in the above minimum weight tour is 1, leading to a total weight of 4, the number of vertices. All edges (of weight 1) in this minimum weight tour also exist in G. Hence, G has a HC.



Graph G

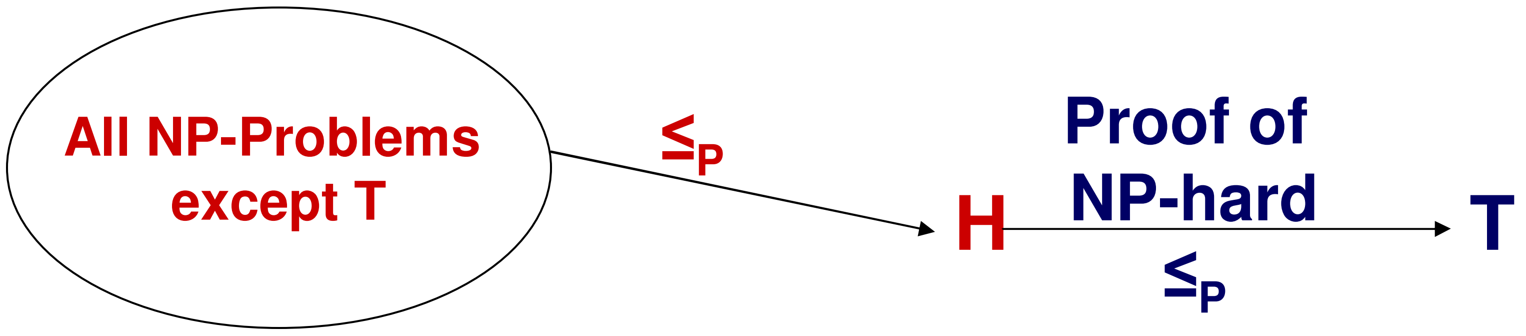


Graph G*

A – C – B – D – A
Any minimum weight tour of G* has to include at least one edge weight of 2, as we cannot find a tour that only involves all edges of weight 1. This implies the original graph G does not have a HC.

# NP-hard Problems

- A problem is said to be NP-hard if every problem in the class NP is polynomial-time reducible to it.
- The TSP problem is a NP-hard problem because every problem in the class NP (like the HC problem) is polynomial-time reducible to it.
- To prove a problem T (like the TSP problem) is NP-hard, we simply take a known NP-hard problem H (like the HC problem that is already proven to be NP-hard) and prove H $\leq_P$ T.

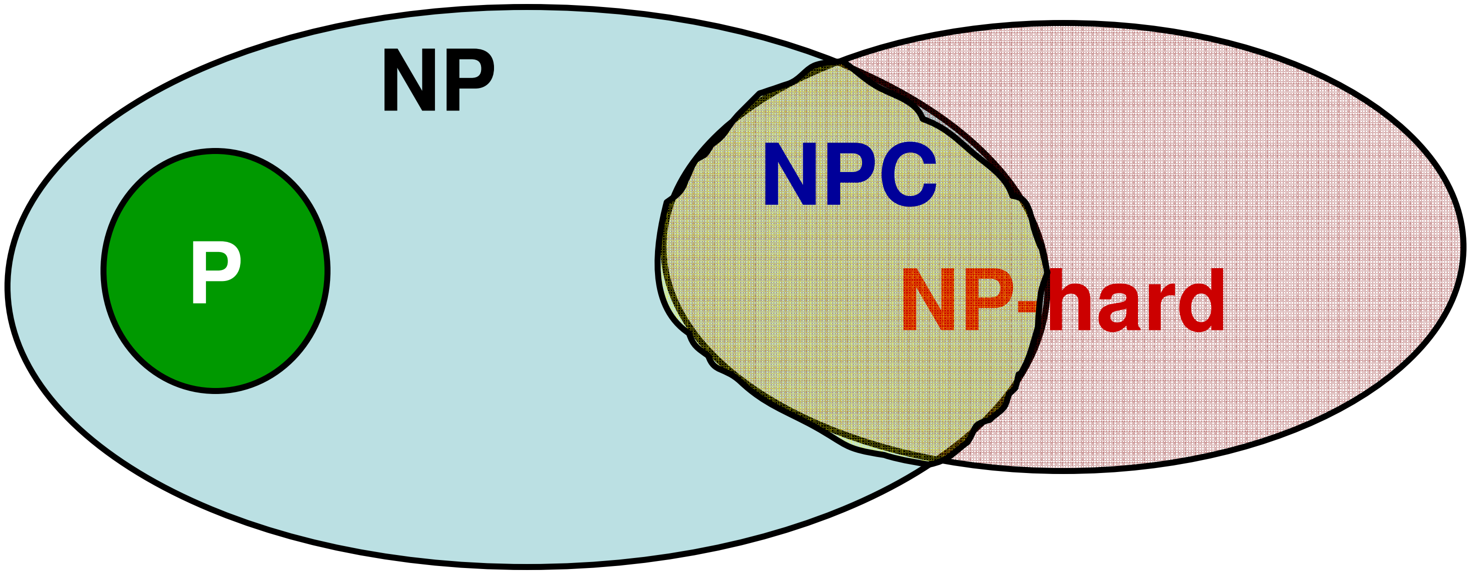


# NP-Complete (NPC) Problems

- A problem X is said to be NP-Complete if:
  - X is in NP
  - X is NP-hard
- If there exists a polynomial time algorithm for any optimization or decision problem (say Y) in the class NP-complete, then every problem X in the class NP can be polynomial-time reducible to Y and solved in polynomial-time. In such a case, P = NP.
- As of now, we do not have a polynomial-time algorithm for any NP-complete problem. But, we are not able to prove that there will never be a polynomial-time algorithm for any NP-complete problem. Hence, it is not clear whether or not P = NP.
- For several NP-complete problems, there exists heuristics (algorithms) that give an approximation solution (not guaranteed to be an optimal solution) in polynomial-time.

# P, NP, NP-hard, NPC

**As of now (and most likely forever!)**
**P ≠ NP**

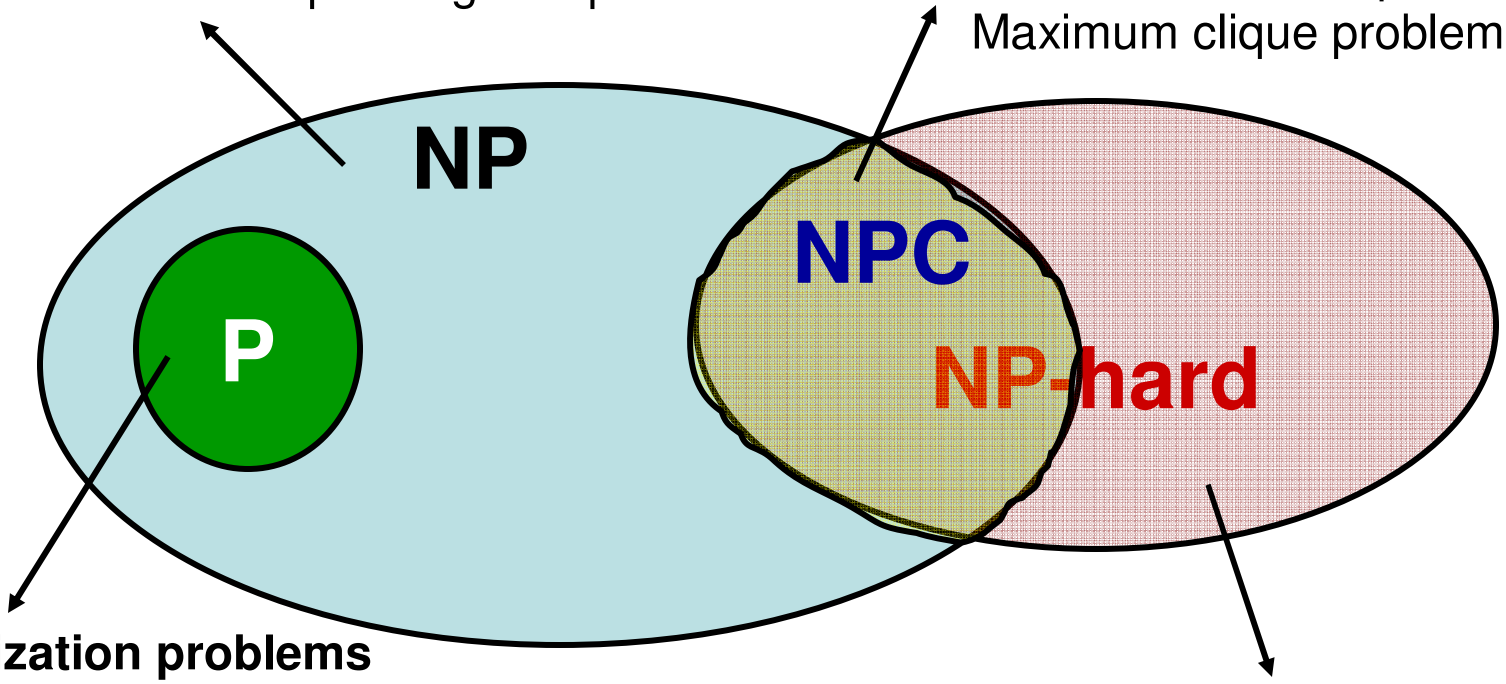


As we can see, there are problems that are NP-hard but not in NP.
One such problem is the halting problem
**Halting Problem:** Given a program and its input, will it run for ever?
This is a decision problem to which every NP-problem can be reduced in Polynomial-time, but there is no way to even guess a solution for this problem.

# P, NP-hard, NPC: Sample Problems

Decision version of the
Minimum spanning tree problem

Decision version of TSP
Hamiltonian circuit problem
Maximum clique problem

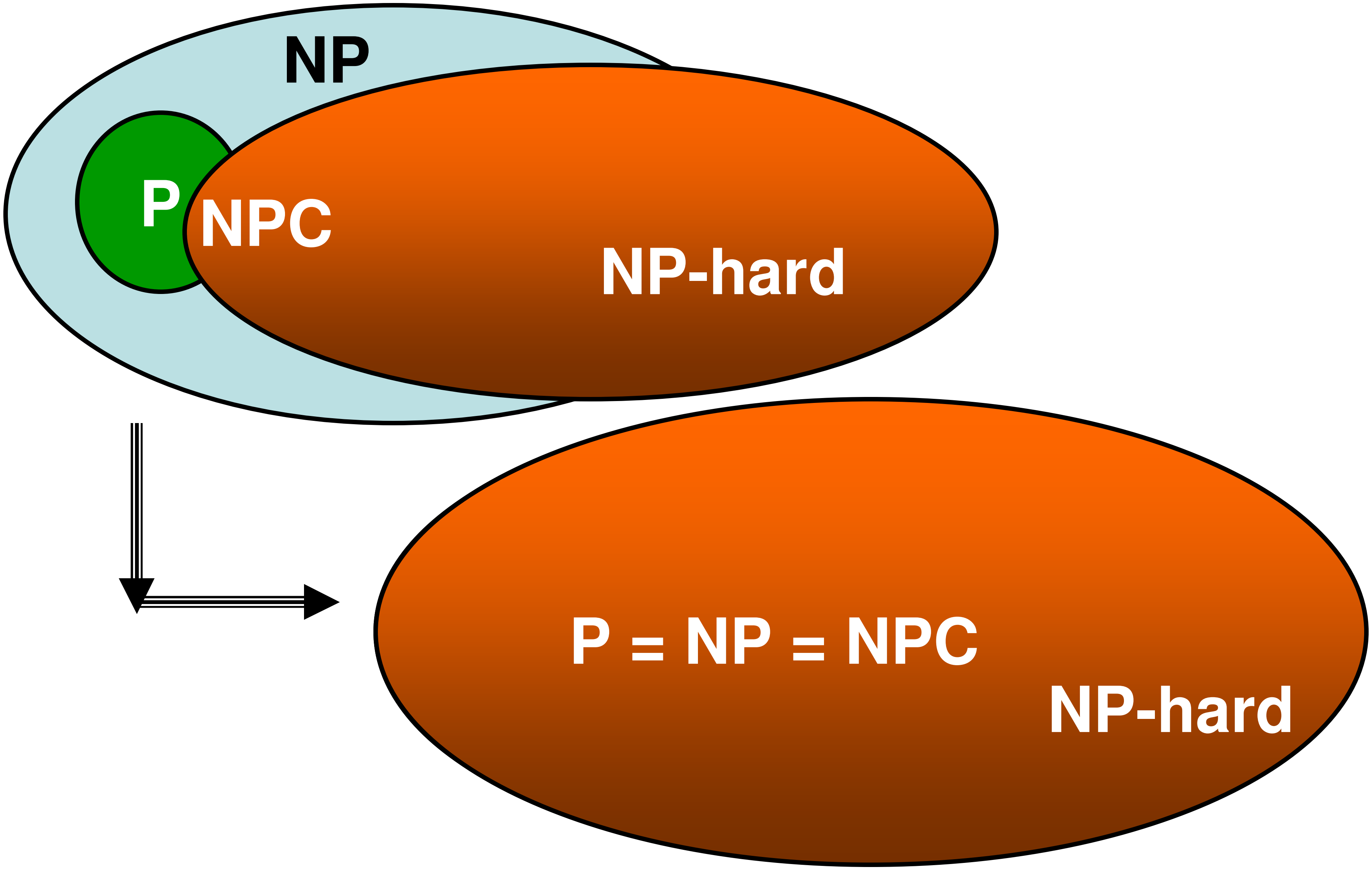


**Optimization problems**
Sorting problem
Minimum spanning tree problem
Shortest path problem

Halting Problem
Optimization version of TSP

**Decision problems**
2-colorability, Is a directed graph a DAG?

# The P $\overset{?}{=}$ NP Dilemma



# Heuristic 1: Nearest Neighbor (NN) Heuristic for the TSP Problem

- Start the tour with a particular vertex, and include it to the tour.
- For every iteration, a vertex (from the set of vertices that are not yet part of the tour) that is closest to the last added vertex to the tour is selected, and included to the tour.
- The above procedure is repeated until all vertices are part of the tour.
- **Time Complexity:** It takes O(V) times to choose a vertex among the candidate vertices for inclusion as the next vertex on the tour. This procedure is repeated for V-1 times. Hence, the time complexity of the heuristic is O($V^2$).

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | 5 | 3 | 8 | 6 | 4 |
| v2 | 5 | 0 | 5 | 4 | 3 | 2 |
| v3 | 3 | 5 | 0 | 5 | 6 | 7 |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | 5 | 6 | 0 |

Initial (pick v1): v1

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | 5 | 3 | 8 | 6 | 4 |
| v2 | 5 | 0 | 5 | 4 | 3 | 2 |
| v3 | 3 | 5 | 0 | 5 | 6 | 7 |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | 5 | 6 | 0 |

# NN Heuristic Example (contd..)

**Iteration 1 (pick v3; 0+3 = 3): v1 - v3**

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | 5 | ~~3~~ | 8 | 6 | 4 |
| v2 | 5 | 0 | 5 | 4 | 3 | 2 |
| v3 | 3 | 5 | 0 | 5 | 6 | 7 |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | 5 | 6 | 0 |

**Iteration 2 (pick v2; 3 +5=8): v1 - v3 - v2**

**Break the tie by choosing the vertex with the lower ID**

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | ~~5~~ | 3 | 8 | 6 | 4 |
| v2 | 5 | 0 | 5 | 4 | 3 | 2 |
| v3 | 3 | ~~5~~ | 0 | 5 | 6 | 7 |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | 5 | 6 | 0 |

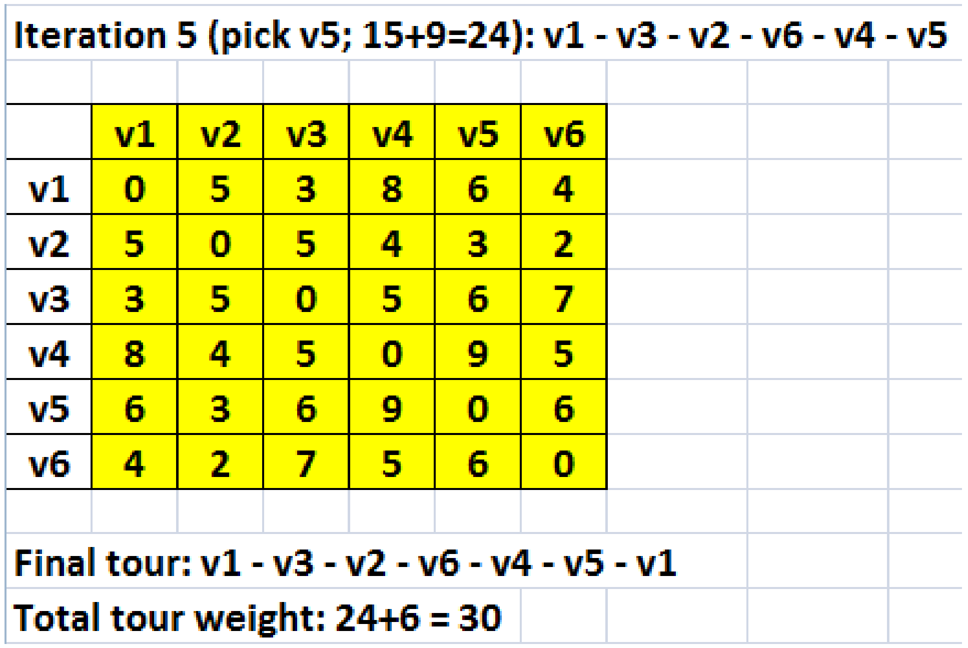
**Iteration 3 (pick v6; 8+2=10): v1 - v3 - v2 - v6**

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | 5 | 3 | 8 | 6 | ~~4~~ |
| v2 | 5 | 0 | 5 | 4 | 3 | ~~2~~ |
| v3 | 3 | 5 | 0 | 5 | 6 | ~~7~~ |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | 5 | 6 | 0 |

**Iteration 4 (pick v4; 10+5=15): v1 - v3 - v2 - v6 - v4**

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | 5 | 3 | ~~8~~ | 6 | 4 |
| v2 | 5 | 0 | 5 | ~~4~~ | 3 | 2 |
| v3 | 3 | 5 | 0 | ~~5~~ | 6 | 7 |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | ~~5~~ | 6 | 0 |

# NN Heuristic Example (contd..)

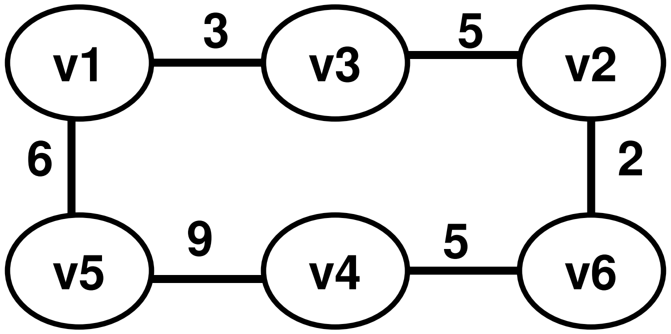


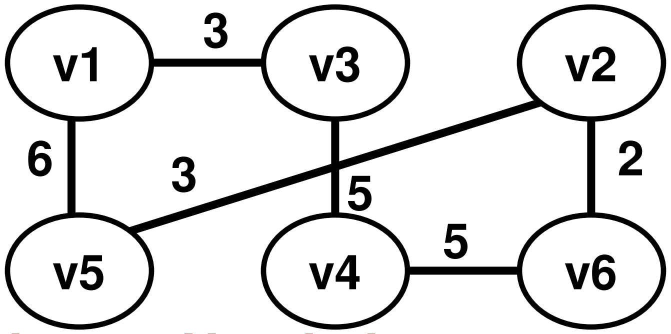
Iteration 5 (pick v5; 15+9=24): v1 - v3 - v2 - v6 - v4 - v5

| | v1 | v2 | v3 | v4 | v5 | v6 |
|---|---|---|---|---|---|---|
| v1 | 0 | 5 | 3 | 8 | 6 | 4 |
| v2 | 5 | 0 | 5 | 4 | 3 | 2 |
| v3 | 3 | 5 | 0 | 5 | 6 | 7 |
| v4 | 8 | 4 | 5 | 0 | 9 | 5 |
| v5 | 6 | 3 | 6 | 9 | 0 | 6 |
| v6 | 4 | 2 | 7 | 5 | 6 | 0 |

Final tour: v1 - v3 - v2 - v6 - v4 - v5 - v1

Total tour weight: 24+6 = 30





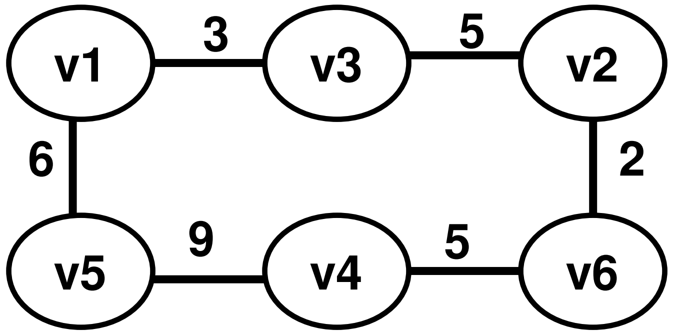
## Improvement to the NN Heuristic using 2-Change Heuristic

Pick two non-overlapping edges (with no common end vertices) and see if we can swap for them using edges that connect the end vertices so that the connectivity of the tour is maintained and the tour cost can be further reduced.

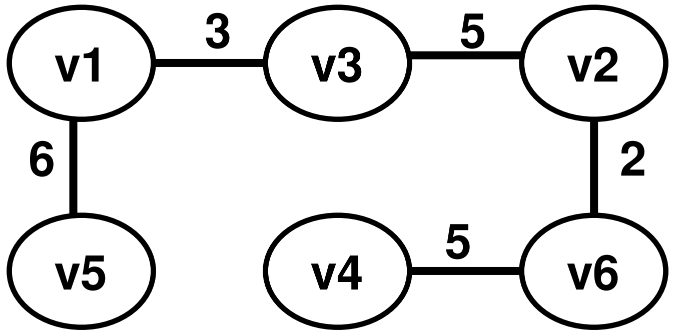
**Strategy:** Pick the costliest edge and a non-overlapping edge (i.e., no common end vertices) that is the next costliest

In the above example, we can pick v5 – v4 (edge wt: 9) and the next costliest non-overlapping edge v3 – v2 (edge wt: 5) and replace them with edges v5 – v2 (wt: 3) and v4 – v3 (wt: 5). The revised tour is v1 – v3 – v4 – v6 – v2 – v5 – v1; tour weight: 24

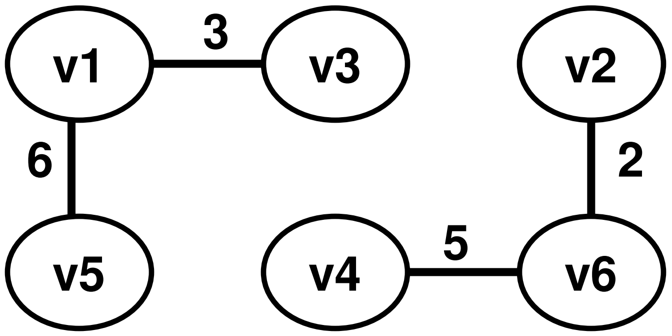
# Execution of the 2-change heuristic



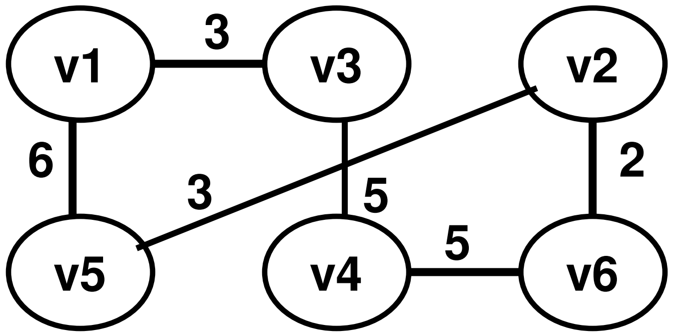
Original Tour



After removing the costliest edge



Remove the next costliest non-overlapping edge is V2 – V3 of weight 5; the only way We can connect the two components and obtain a tour is to connect V2 with V5 (V2 – V5) and V3 with V4 (V3 – V4).

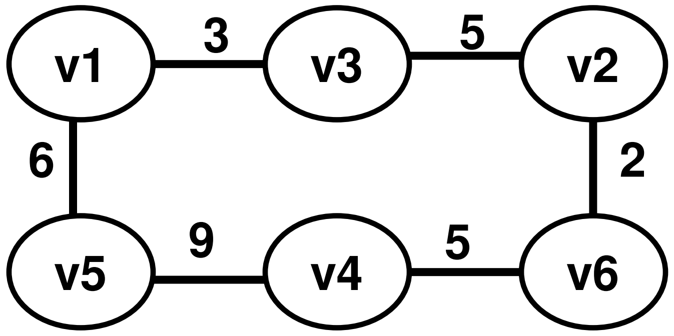


Modified Tour

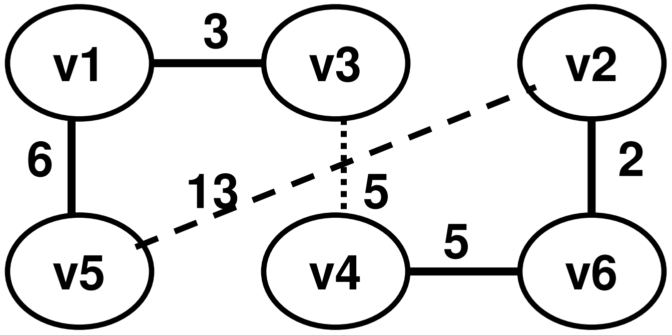
We stay with the modified tour since the sum of the weights of the edges added (3 + 5 = 8) is lower than the sum of the edge weights removed (9 + 5 = 14)

# 2-Change Heuristic (may not always work)

- Note that 2-change heuristic cannot guarantee to improve (i.e., reduce) the weight of a tour.
- The improvement really depends on the weights and locations of the two edges that we remove and the two that we add.
  - So, it essentially depends on the graph.
- If the 2-change heuristic does not improve the tour weight, we stay with the original tour determined by the TSP heuristic.



Original tour weight = 30



If the weight of edge V2 – V5 is 13 (instead of 3), then the weight of the modified tour is 30 – (9 + 5) + (13 + 5) = 34 > 30. So, we stay with the original tour of weight 30.

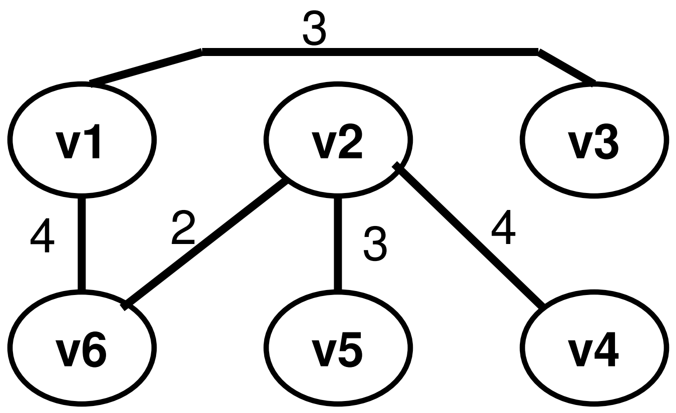
# Heuristic # 2 for the TSP Problem Twice-around-the Tree Algorithm

- **Step 1:** Construct a Minimum Spanning Tree of the graph corresponding to a given instance of the TSP problem
- **Step 2:** Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by. This can be done by a DFS traversal.
- **Step 3**: Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. The vertices remaining on the list will form a Hamiltonian Circuit, which is the output of the algorithm.

**Note:** We will use the **principle of Triangle Inequality** for Euclidean plane:
The sum of the two sides of a triangle is greater than the third side of the triangle

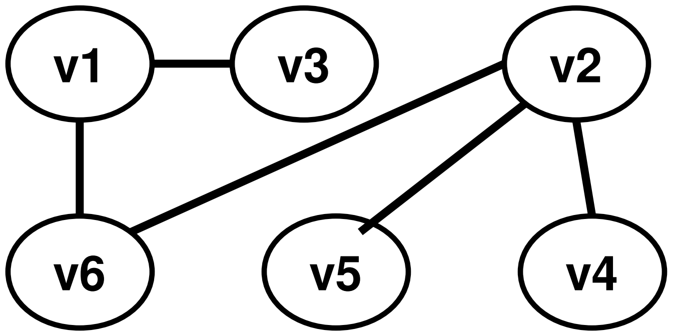
| edge wt | edge | | edge wt | edge |
|---|---|---|---|---|
| 2 | v2 - v6 | | 5 | v4 - v6 |
| 3 | v1 - v3 | | 6 | v1 - v5 |
| 3 | v2 - v5 | | 6 | v3 - v5 |
| 4 | v1 - v6 | | 6 | v5 - v6 |
| 4 | v2 - v4 | | 7 | v3 - v6 |
| 5 | v1 - v2 | | 8 | v1 - v4 |
| 5 | v2 - v3 | | 9 | v4 - v5 |
| 5 | v3 - v4 | | | |

**Step 1: MST of the Graph**



# Heuristic # 2 for the TSP Problem Twice-around-the Tree Algorithm

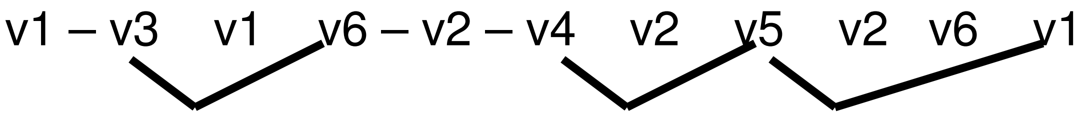
**MST (vertices rearranged) from Step 1**



**Step 3: Optimizing the DFS Walk**

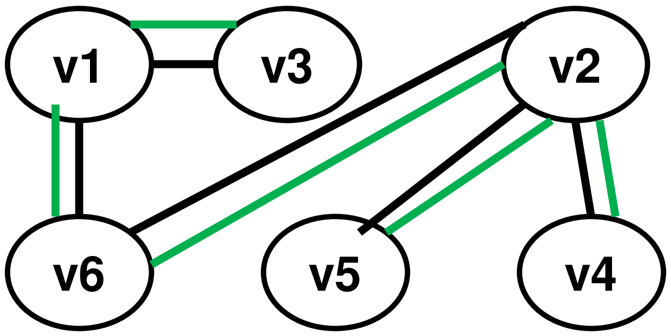
**Tour from Step 2:**
v1 – v3 – v1 – v6 – v2 – v4 – v2 – v5 – v2 – v6 – v1

**Optimized Tour:**

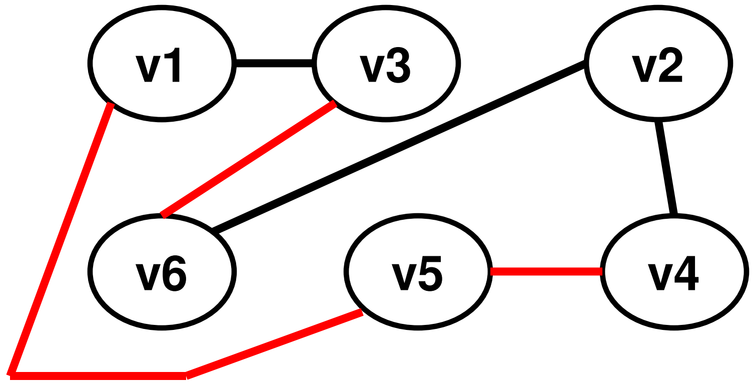


**Step 2: DFS Traversal of the MST**

v1 – v3 – v1 – v6 – v2 – v4 – v2 – v5 – v2 – v6 – v1



v1 – v3 – v6 – v2 – v4 – v5 – v1
Tour Weight: 31

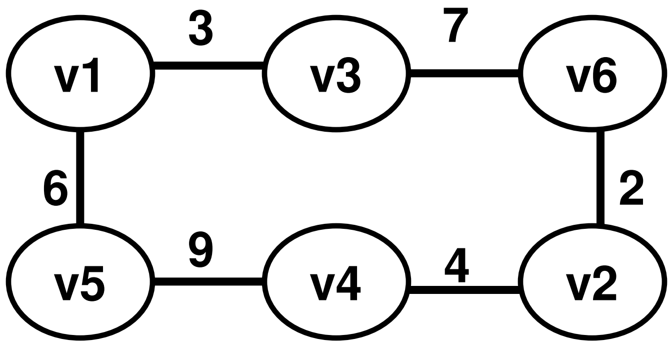


# Heuristic # 2 for the TSP Problem Twice-around-the Tree Algorithm

**TSP Tour of Twice-around-the-Tree Algorithm**

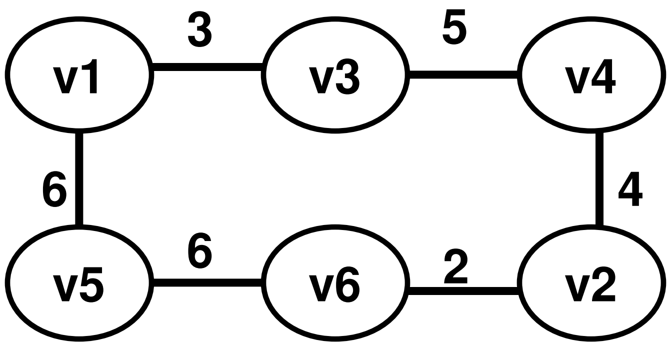
| edge wt | edge | | edge wt | edge |
|---|---|---|---|---|
| 2 | v2 - v6 | | 5 | v4 - v6 |
| 3 | v1 - v3 | | 6 | v1 - v5 |
| 3 | v2 - v5 | | 6 | v3 - v5 |
| 4 | v1 - v6 | | 6 | v5 - v6 |
| 4 | v2 - v4 | | 7 | v3 - v6 |
| 5 | v1 - v2 | | 8 | v1 - v4 |
| 5 | v2 - v3 | | 9 | v4 - v5 |
| 5 | v3 - v4 | | | |

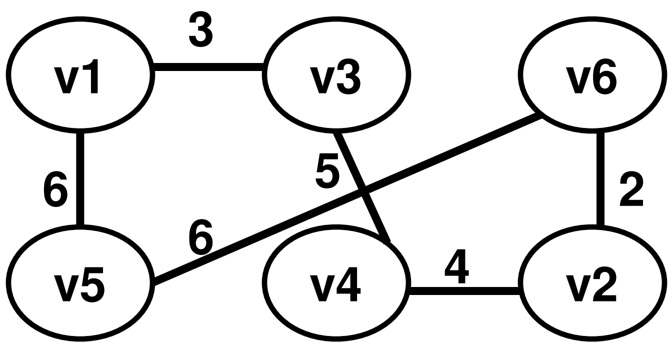
**v1 – v3 – v6 – v2 – v4 – v5 – v1**
**Tour Weight: 31**



**Improved Tour Weight: 26**

**Improvement using 2-Change**





# Proof for the Approximation Ratio for Twice-around-the-Tree

- Step 1: Let wt(MST) be the weight of the MST
- Step 2: wt(MST-Walk) = 2*wt(MST)
- Step 3: wt(heuristic tour) ≤ wt(MST-Walk)
- We also know that wt(MST) < wt(Optimal tour)
  - i.e., an MST of ‘n’ vertices should weigh less than an optimal tour of the ‘n’ vertices.
- Substitute for wt(MST-Walk) in the result of Step 3.
  - Wt(heuristic tour) ≤ 2*wt(MST)
  - Wt(heuristic tour) < 2*wt(Optimal tour).
    - The factor of ‘2’ is the approximation ratio.

**Application of this Theorem:** **In the previous example, we saw that the Twice-around-the-tree (before the improvement with the 2-change heuristic) gave us a tour of Weight 31. Applying the theorem, we can say that the weight of the optimal tour is greater than 31/2 = 15.5. Since the edge weights are all integers, we can say the Weight of the optimal tour for the given graph is at least 16.**

## Proof for the Approximation Ratio for Twice-around-the-Tree

- Let wt(MST) be the weight of the MST generated from Step 1.
- The weight of the DFS walk generated from Step 2 could be at most 2*w(MST), as seen in the example.
- In Step 3, we are trying to optimize the DFS walk and extract a Hamiltonian Circuit of lower weight. Even if no optimization is possible, the weight of the tour generated by the Twice-around-the-Tree algorithm is at most twice the weight of the minimum spanning tree of the graph instance of the TSP problem.
- Note that w(MST) of the graph has to be less than the weight of an optimal tour, w(Optimal Tour). Otherwise, if w(Optimal Tour) ≤ w(MST), then the so-called MST with V-1 edges is not a MST.
- W(Twice-around-the-Tree tour) ≤ 2*W(MST) < 2*w(Optimal Tour).
- W(Twice-around-the-Tree tour) / W(Optimal Tour) < 2.
- Hence, the approximation ratio of the Twice-around-the-Tree algorithm is less than 2.

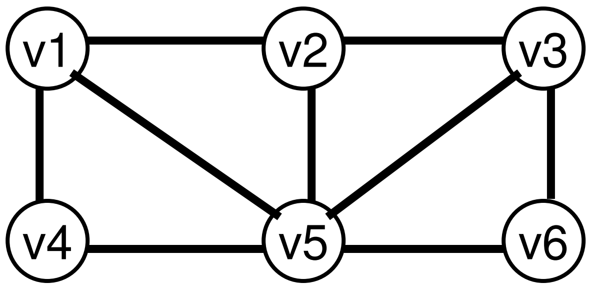
## Independent Set, Vertex Cover, Clique

- An independent set of a graph *G* is a subset *IS* of vertices such that there is no edge in G between any two vertices in IS.
- Optimization Problem: Find a maximal independent set of a graph
- Decision Problem: Given a graph G, is there an independent set of G of size k?
  - The objective is to find a subset of k vertices from the set of vertices in G, such that any two vertices among the k vertices do not have an edge in G.

- A Vertex Cover for a graph G is a subset of vertices VC such that every edge in G has at least one end vertex in VC.
- The optimization problem is to find the vertex cover with the minimal set of vertices.
- Note that VC = V – IS, where V is the set of vertices and IS is the Independent Set.

- A Clique of a graph G is a subset C of vertices such that there is an edge in G between any two vertices in C.
- Similar to the Independent set problem, one can have optimization and decision versions of the Clique problem. The objective will be to find a maximum clique of k vertices or more.

# Independent Set, Vertex Cover, Clique



In the above graph,
the set of vertices
- {v2, v4, v6} form an Independent Set
- Thus, {v1, v3, v5} form the Vertex Cover

- {v1, v2, v5} form a Clique

Polynomial Reduction:

Given a graph G, find a Complement graph G* containing the same set of vertices, such that there exists an edge between any two vertices in G* if and only if there is no edge between them in G.

*An Independent Set in G* is a Clique in G and vice-versa*: the only reason there is no edge between two vertices in G* is because there is an edge between them in G.

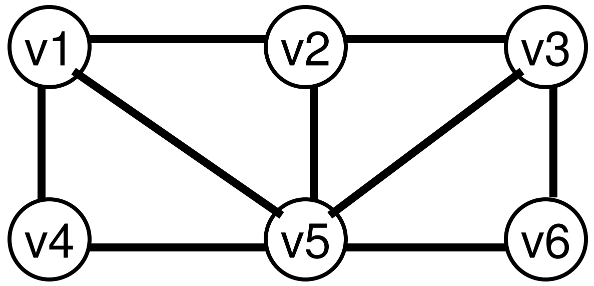
Approach to find the Independent Set, Vertex Cover and Clique for a Graph G

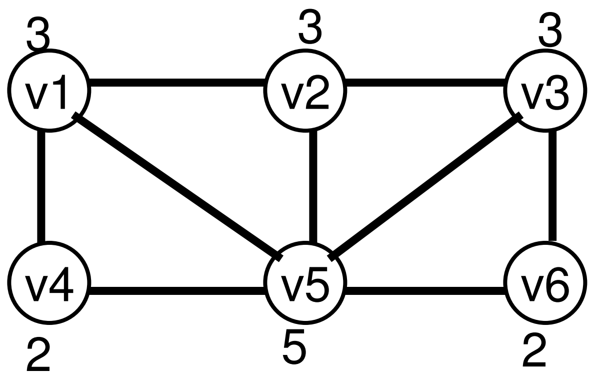
1. Find the Independent Set, IS, of graph G using the Minimum Neighbors Heuristic
2. The Vertex Cover of G is V – IS, where V is the set of vertices in G
3. Find the Complement Graph G* of G and run the Minimum Neighbors Heuristic on it. The Independent Set of G* is the Clique of G.

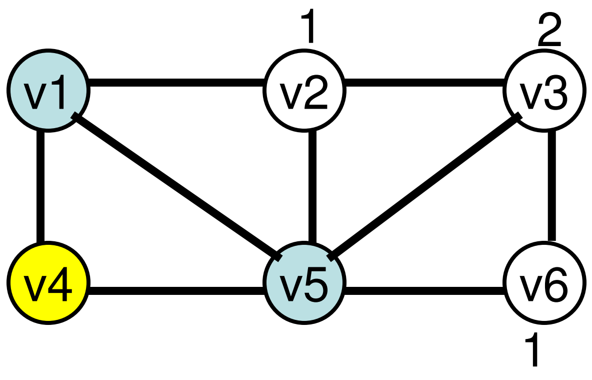
# Proof for the Polynomial Reductions

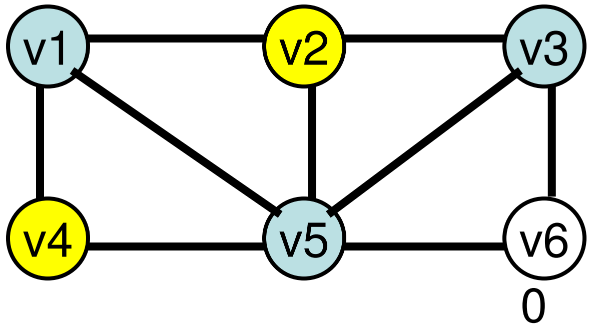
- **Clique $\leq_P$ Independent Set**
- Let G* be the complement graph of G. There exists an edge in G* if and only if there is no edge in G.
- Determine a Maximal Independent Set C* of G*. There exists no edge between any two vertices in C* of G*. ==> There exists an edge between the two vertices of C* in G. For any two vertices in C*, there is an edge in G. Hence, C* is a Maximal Clique of G.
- **Independent Set $\leq_P$ Clique**
- By the same argument as above, we can determine a maximal clique in G*; there is an edge between any two vertices in the maximal clique of G* ➔there are no edges between any two of these vertices in G. These vertices form the maximal independent set in G.
- **Vertex Cover $\leq_P$ Independent Set**
- Let IS be an independent set of graph G.
- Let the vertex cover of G be V – IS, where V is the set of all vertices in the graph.
- For every edge in G, the two end vertices are not in IS. Hence, at least one of the two end vertices must be in V – IS. Thus, V – IS should be a vertex cover for G.

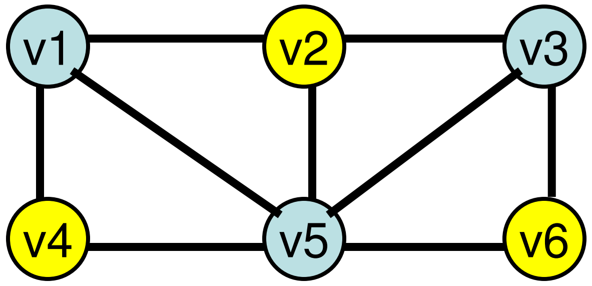
# Example 1 to Find Independent Set using the Minimum Neighbors Heuristic











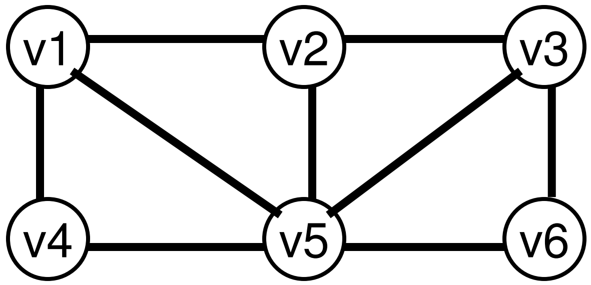
Idea: Give preference to vertices with minimal number of (uncovered) neighbors to be part of the Independent Set. A vertex is said to be covered if itself or any of its neighbors in the Independent Set.

Independent Set for the above graph = {v2, v4, v6}

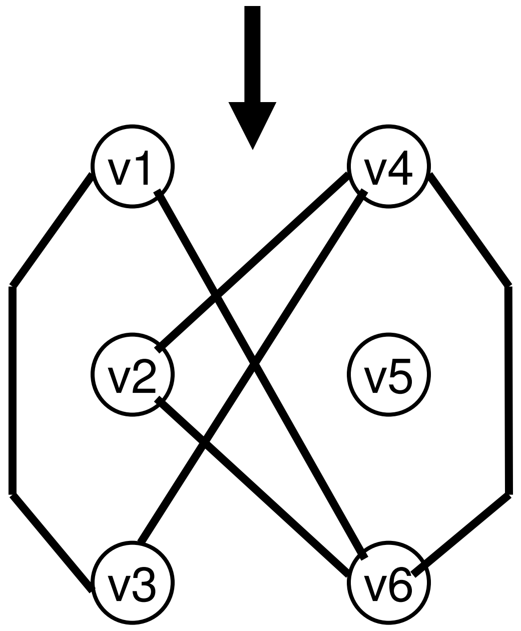
This is also the Maximal Independent Set (i.e., there exists no Independent Set of size 4 or more for the above graph). However, the heuristic is not guaranteed in general to give a maximal Independent set.

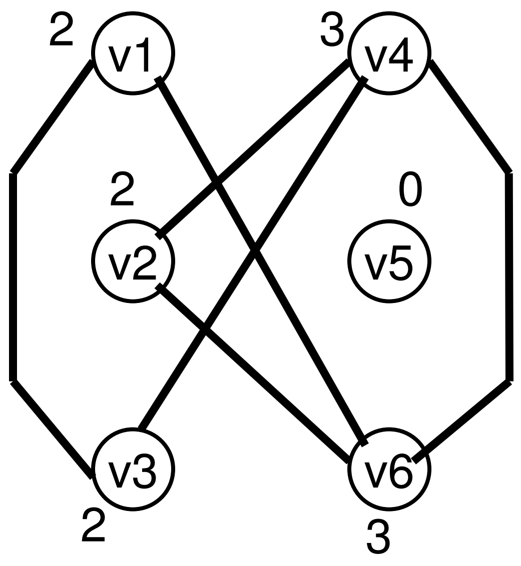
Vertex Cover = {v1, v3, v5}

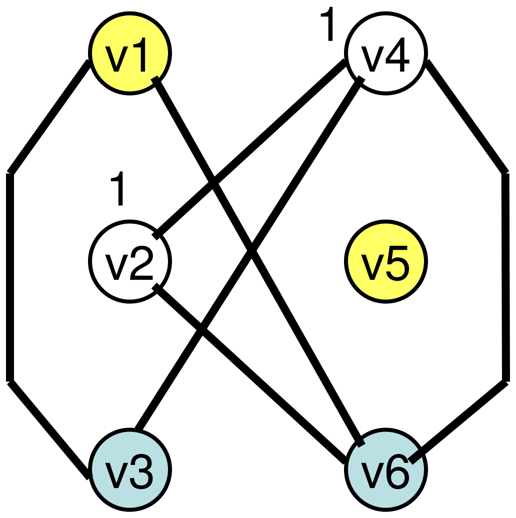
Given G -------->



Find G*, complement of G

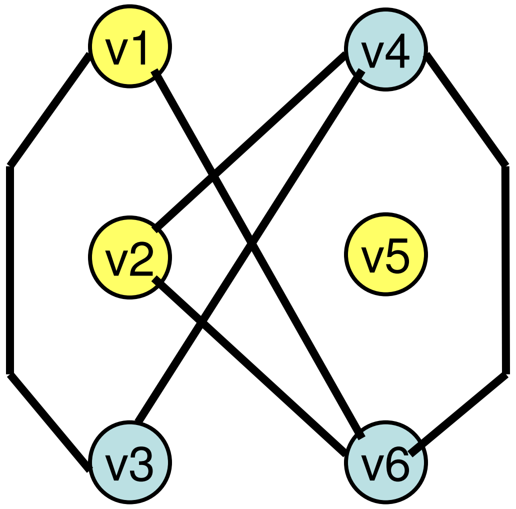




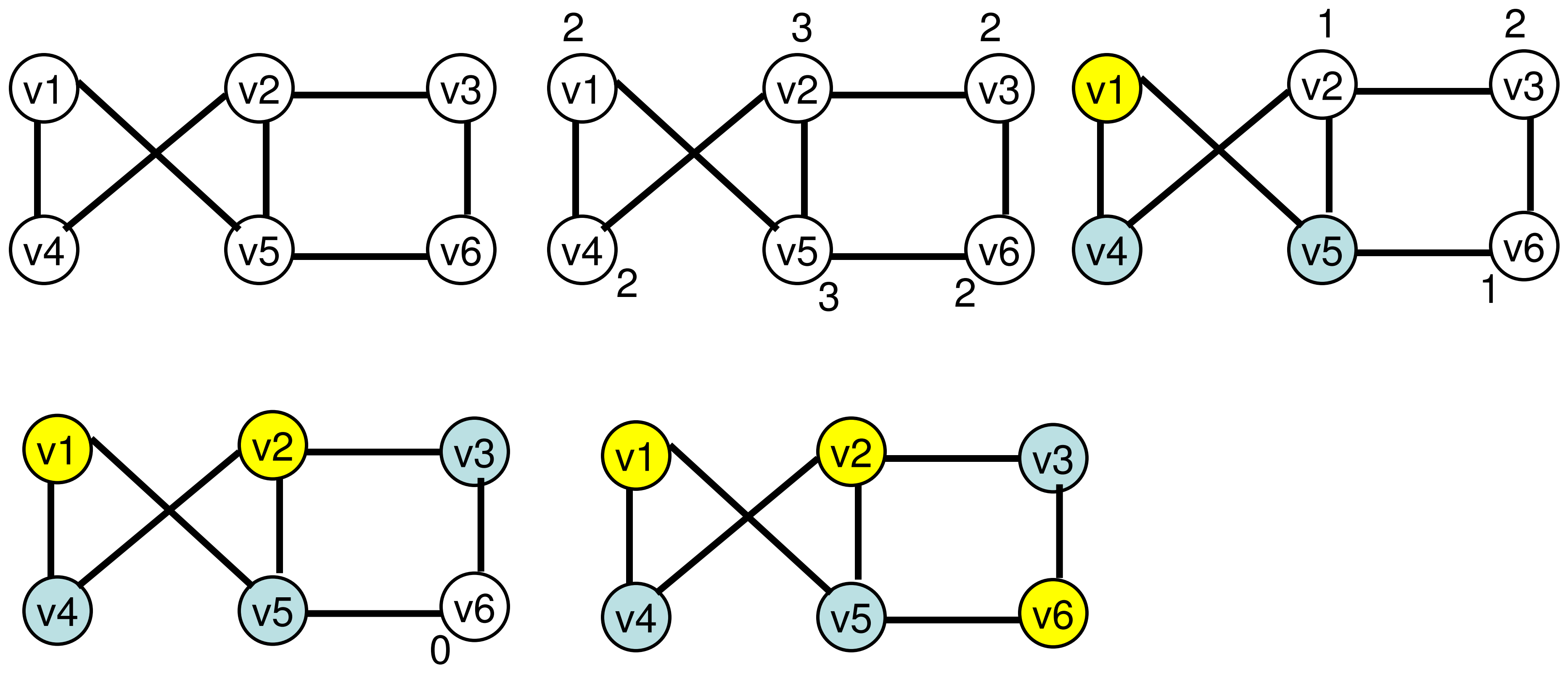


## Example 1 to Determine a Clique Using the Minimum Neighbors Heuristic to Approximate an Independent Set

{v1, v2, v5} is an Independent Set in G* and it is a clique in G.



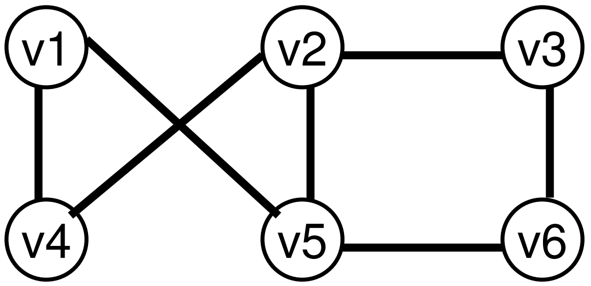
# Example 2 to Find Independent Set using the Minimum Neighbors Heuristic



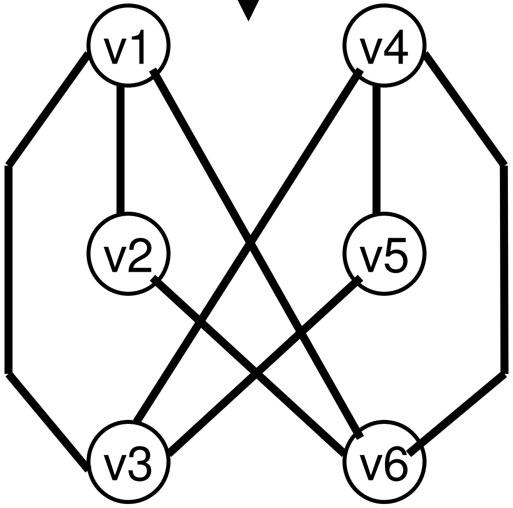
Independent Set = {v1, v2, v6}

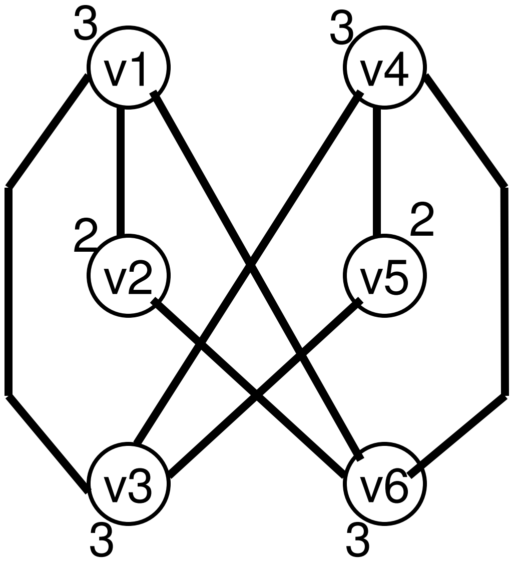
Vertex Cover = {v3, v4, v5}

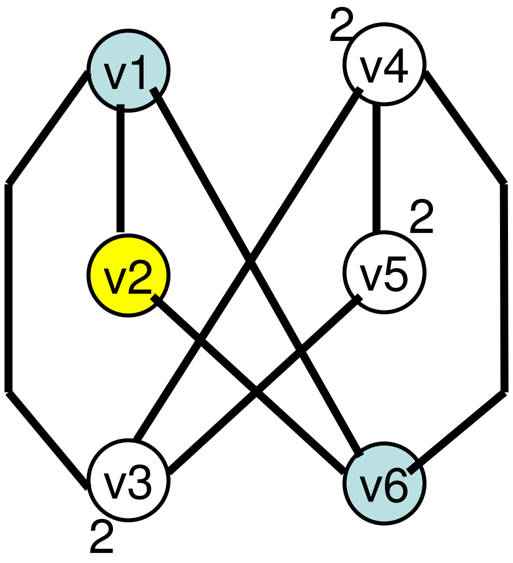
Given G ------->

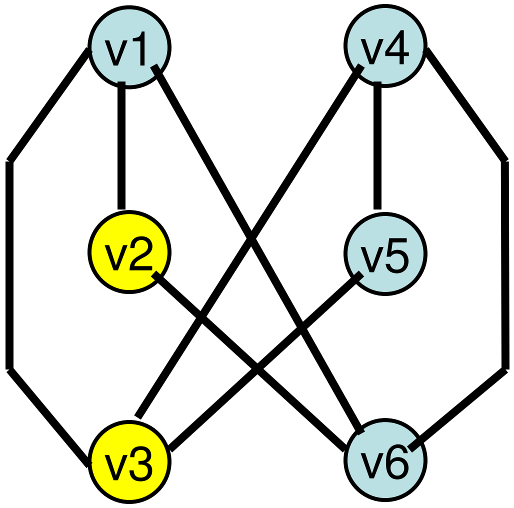


Find G*, complement of G







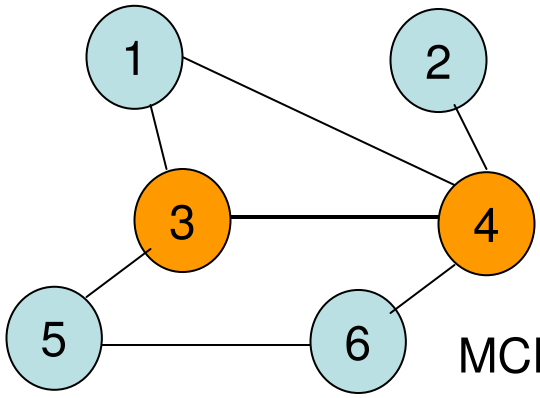


Independent Set of G* = {v2, v3}
Clique of G = {v2, v3}

Note that Clique of G* = {v1, v2, v6} is an Independent Set of G, leading to a Vertex Cover of {v3, v4, v5} of G.

# Minimum Connected Dominating Set

- Given a connected undirected graph G = (V, E) where V is the set of vertices and E is the set of edges, a connected dominating set (CDS) is a sub-graph of G such that all nodes in the graph are included in the CDS or directly attached to a node in the CDS.
- A minimum connected dominating set (MCDS) is the smallest CDS (in terms of the number of nodes in the CDS) for the entire graph.
- For broadcast communication, it is sufficient if the data goes through all the nodes in the MCDS. Each node in the MCDS can in turn forward the data to its neighbors.
- Determining the MCDS in an undirected graph is NP-complete.
- Degree of a vertex is the number of neighbors of the vertex



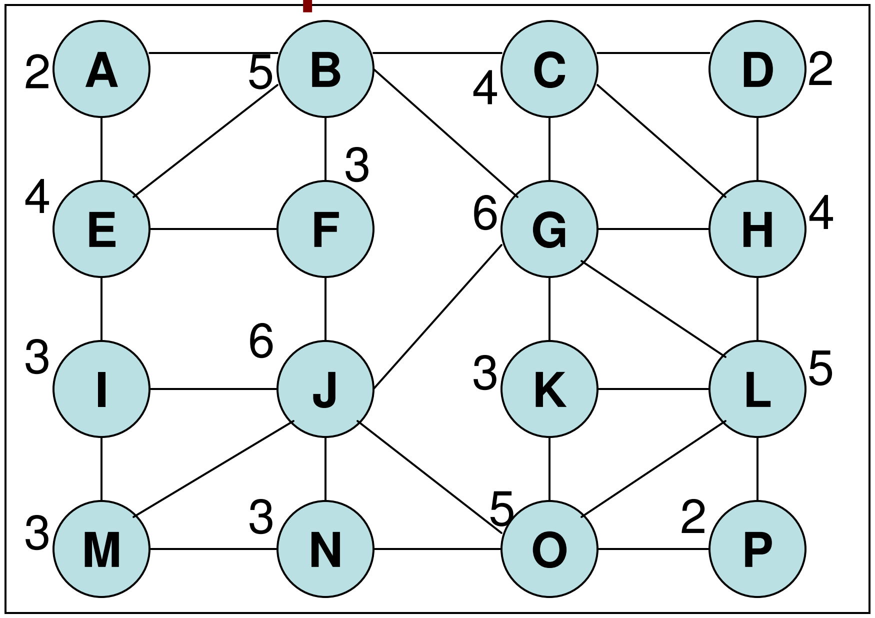
MCDS = [3, 4]

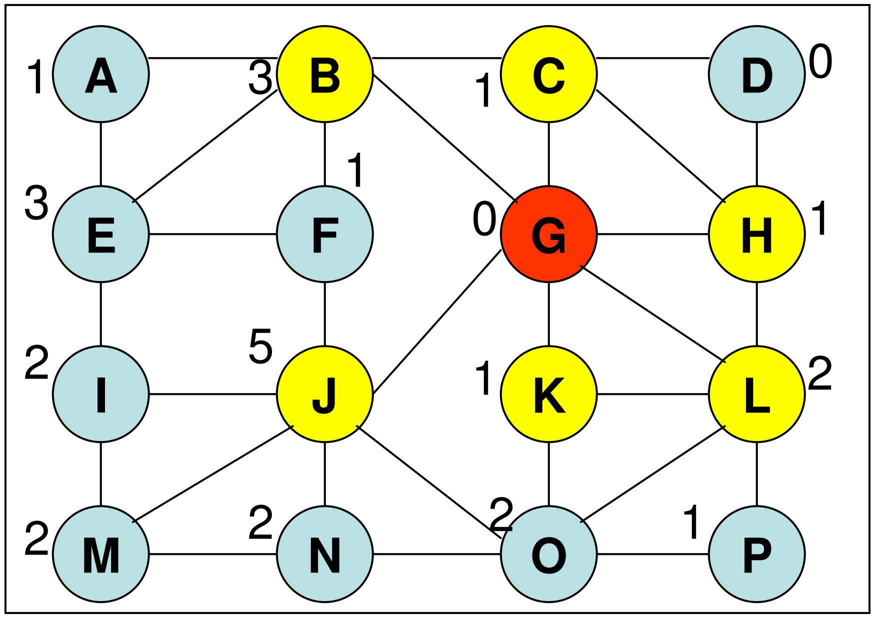
The size of a MCDS clearly depends on the degree of the nodes. Hence, we will study a degree-based heuristic to approximate the MCDS

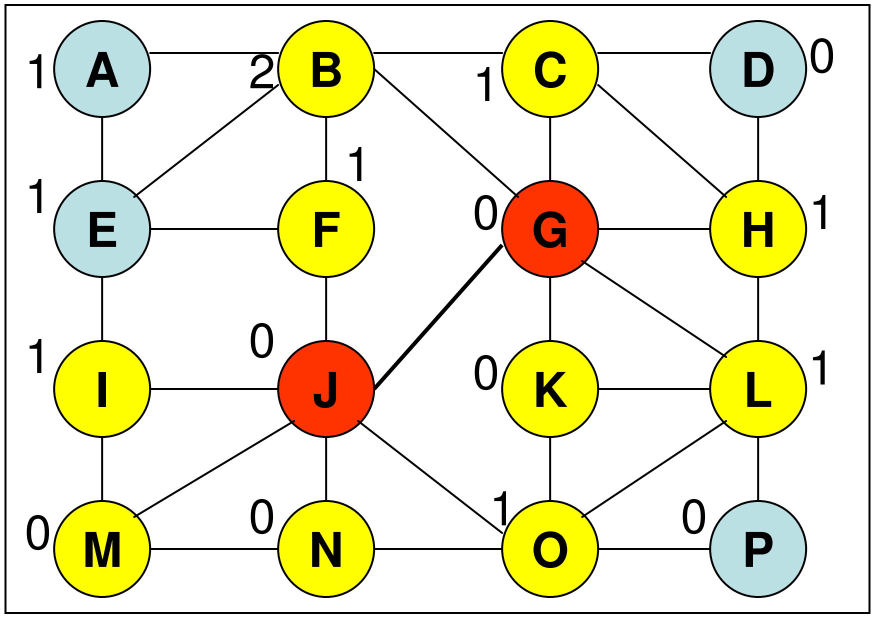
# Heuristic to Approximate a MCDS

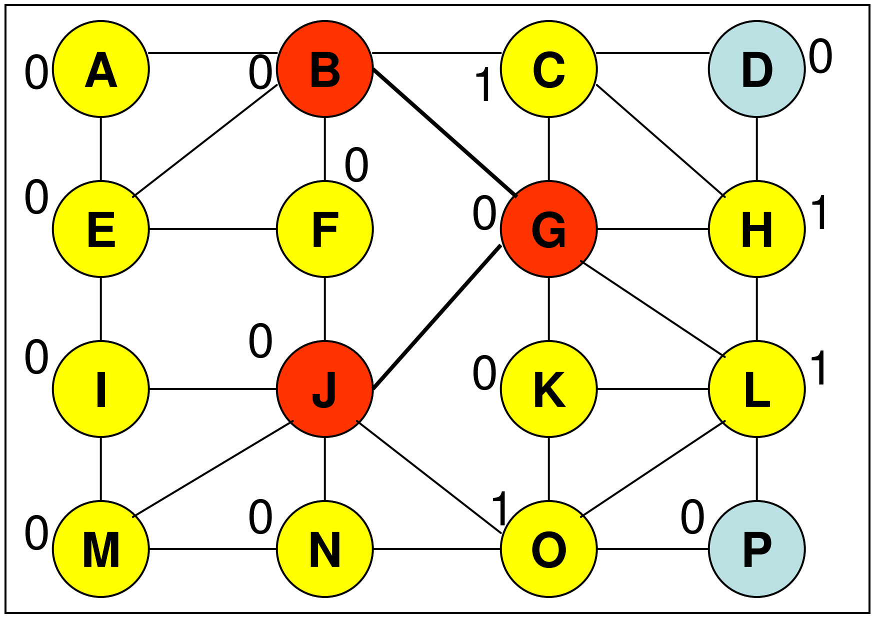
```
Input: Graph $G = (V, E)$, where $V$ is the vertex set and $E$ is the edge set.
          Source – vertex, $s$  $V$.
Auxiliary Variables and Functions:
          CDS-list, Covered-list, Neighbors($v$) for every $v$ in $V$.
Output: CDS-list
Initialization: Covered-list = {$s$}, CDS-list = $\Phi$
Begin d-MCDS
  while ( |Covered-list| < |V| ) do
    Select a vertex $r \in$ Covered-list and $r \notin$ CDS-list such that $r$ has the
    maximum neighbors that are not in Covered-list.
    CDS-list = CDS-list U {$r$}
        For all $u \in$ Neighbors($r$) and $u \notin$ Covered-list,
            Covered-list = Covered-list U {$u$}
end while
return CDS-list
End d-MCDS
```

# Example for *d-MCDS* Heuristic

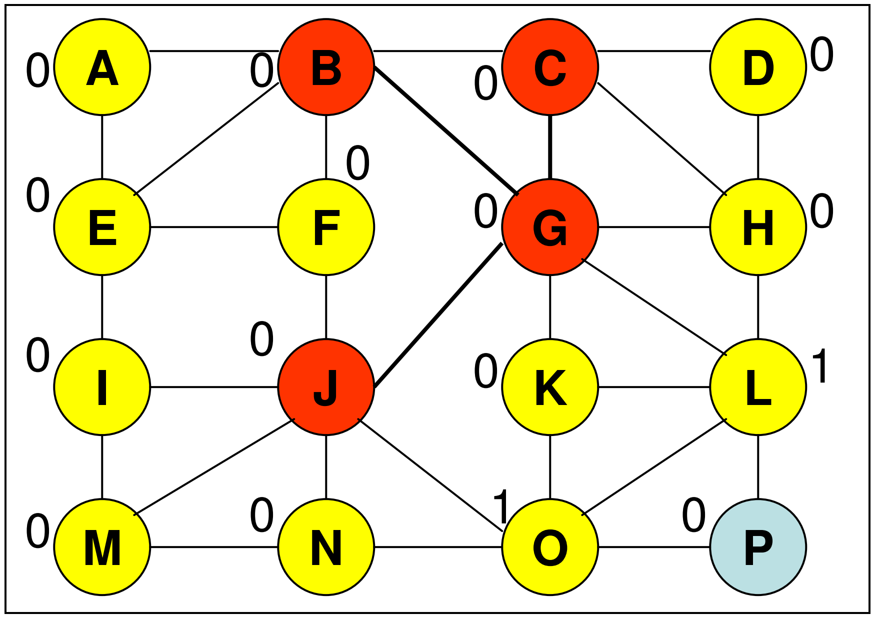


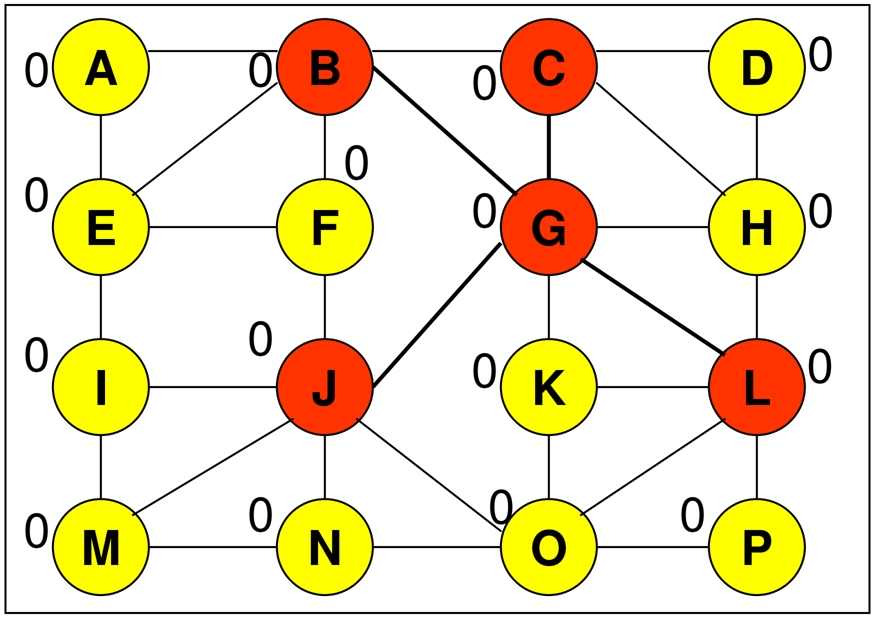


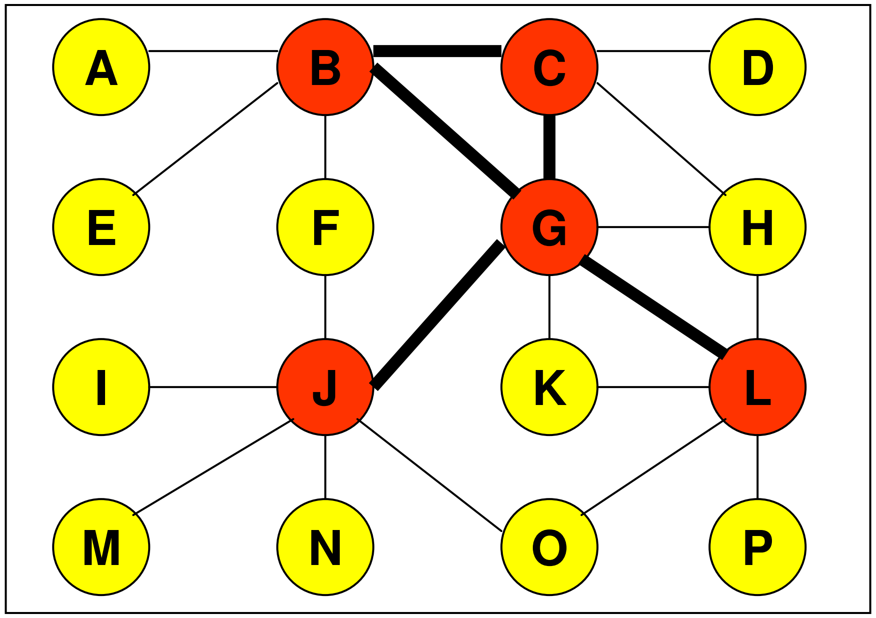




# Example for *d-MCDS* Heuristic







MCDS Nodes = [G, J, B, C, L]