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

P, NP-Problems

- Class P: A class of optimization (min/max solutions) or decision problems (yes/no solutions) for which there exists algorithms to solve them with a worst-case time complexity of O(p(*n*)) where p(*n*) is a polynomial (incl. log time) of the problem's input size *n*.

- Note that there are many decision or optimization problems for which no polynomial-time algorithm has been found; but, neither the impossibility of such an algorithm been proved.
  - Example: Traveling Salesman problem, Hamiltonian Circuit problem

- Hamiltonian Circuit Problem: Determine whether a given graph has a Hamiltonian Circuit – a path that starts and ends at the same vertex and passes through all the other vertices exactly once.
- Traveling Salesman Problem: Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian Circuit in a complete graph with positive integer weights).

P, NP-Problems

- A deterministic algorithm is the one that produces the same output for a problem instance each time the algorithm is run.
- A non-deterministic algorithm is a two-stage procedure that takes as its input an instance *I* of a decision problem and does the following:
  - Stage 1: Non-deterministic ("Guessing") Stage: An arbitrary string S is generated that can be thought of as a candidate solution to the given instance I (could be sometimes, not a correct solution too).
  - Stage 2: Deterministic ("Verification") Stage: A deterministic algorithm takes both I and S as its input and outputs *yes* if S represents a solution to instance I. Note that we do not want a non-deterministic algorithm to output a yes answer on an instance for which the answer should be no.
- A non-deterministic algorithm for the Traveling Salesman problem could be the one that inputs a complete weighted graph of n vertices, then (stage 1): generates an arbitrary sequence of n vertices and (stage 2): verifies whether or not that each vertex, except the starting vertex, is repeated only once in the sequence, and outputs a yes/no answer accordingly.

P, NP-Problems

- Class NP (Non-deterministic Polynomial) is the class of decision problems that can be solved by non-deterministic polynomial algorithms.

- Note that all problems in Class P are in Class NP: We can replace the non-deterministic guessing of Stage 1 with the deterministic algorithm for the decision problem, and then in Stage 2, deterministically verify the correctness of the solution of the deterministic algorithm. In other words, $P \subseteq NP$. However, it is not clear whether $P = NP$ or $P \neq NP$.

- Polynomial reduction: A problem *I* is said to be reducible to another problem *J* in polynomial time (denoted $I \leq_P J$), if every instance of *I* can be reduced to an instance of *J* in polynomial time. So, if a polynomial time algorithm exists to solve an instance of problem *J*, then we can say that there exists a polynomial time algorithm for *I*.

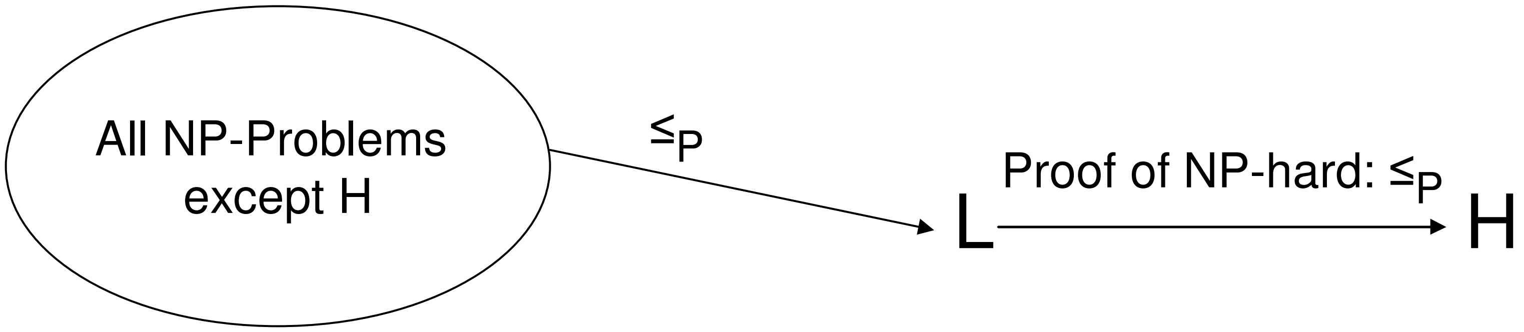
NP-Complete Problems

- **A problem *I* is said to be NP-complete, if:**
  - *I* is in NP
  - Every problem in class NP is polynomial-time reducible to *I*.
- There are several decision problems that have been proven to be NP-complete.
- If we could find a polynomial-time deterministic algorithm to solve any single NP-complete problem, then all problems in NP can be considered to also be solvable deterministically in polynomial-time. In that case P = NP.
  - After proving that the problem they want to solve is an NP-complete problem, researchers do not break their heads to find a polynomial-time deterministic algorithm – if they find one, they would surely win the Turing award though!!!
  - Instead, the focus will be on developing an approximate algorithm (called a heuristic) that can yield a solution that will be bounded below or above (depending on the context) by the optimal solution within a certain value, that is as minimal as possible. That is, aim for an approximation ratio that is closer to 1; but not equal to 1.

NP-Complete Problems

- How would you prove that a problem H is NP-complete?

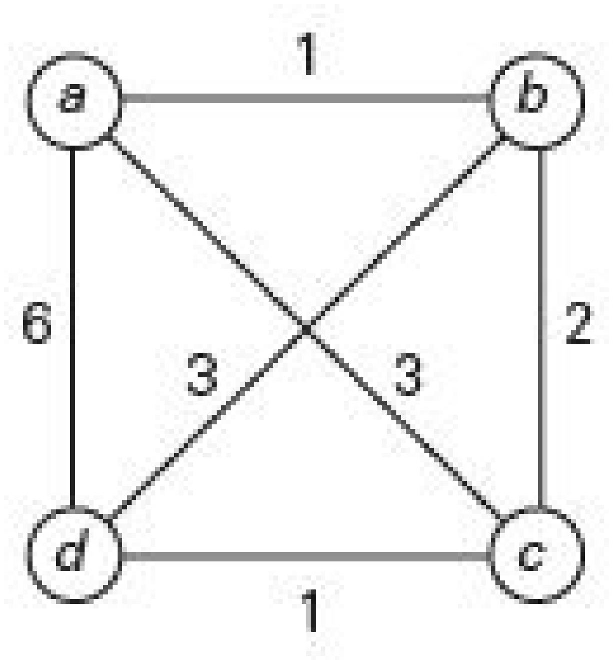
- Let L be a problem that has been already proven to be NP-complete.
- We will show the following:
- (1) H is in NP: There is a polynomial-time algorithm that can verify whether a possible solution given by a non-deterministic algorithm is indeed a solution or not.
- (2) H is NP-hard: A problem H in NP is said to be NP-hard, if a NP-complete problem is polynomial time reducible to H. In this case, we show that $L \leq_P H$.



Traveling Salesman Problem

- Given a complete weighted graph (i.e., weighted edges representing distances in the Euclidean plane between any two vertices), find a minimum weight tour that starts from a particular vertex, goes through all the other vertices exactly once, and ends at the starting vertex.

Traveling Salesman Problem (TSP)



- Possible Combinations and their Tour Weights:

• a – b – c – d – a	10
• a – b – d – c – a	8
• a – c – b – d – a	14
• a – c – d – b – a	8
• a – d – c – b – a	10
• a – d – b – c – a	14

- Similarly 18 more tours, can be written (6 each starting from b, c, or d).
- The minimal tour weight is 8 and it could be one of the tours of the corresponding weight , as listed above.

Traveling Salesman Problem

- Given a complete weighted graph *G* of ‘*n*’ distinct vertices
- Let there be a non-deterministic algorithm that generates a sequence of *n*+1 vertices of *G*.
- **NP: Polynomial-time Verification Algorithm:**
  - Check if the beginning and ending vertices of the sequence are the same.
  - Check if every other vertex in G appears exactly once in the sequence.
  - If a sequence satisfies the above two checks, then it is a solution for the TSP problem; otherwise, not.
- Note that while proving that a problem is in NP, we do not worry about an optimal solution for it (any solution is fine); while proving the problem is NP-hard, we talk about using an optimal solution for the problem to solve a known NP-complete problem.

Traveling Salesman Problem

- Both the Hamiltonian Circuit problem (HCP) and the Traveling Salesman problems (TSP) are NP-Complete problems.
- We will prove the TSP problem is NP-hard by describing a polynomial-time reduction for the Hamiltonian Circuit Problem to a TSP problem.

- **NP-Hard: Polynomial Reduction:**
- We will see the polynomial-time reduction of a Hamiltonian Circuit problem to a Traveling Salesman problem.
- Recollect the Hamiltonian Circuit problem is to find a tour of a given unweighted graph that simply starts at one vertex and goes through all the other vertices and ends at the starting vertex.
- Note that the input graph G to a Hamiltonian Circuit problem need not be a complete graph connecting all vertices.
- For the reduction, we will construct a complete graph G* such that there exists an edge of weight 1 in G* if the edge exists in G; otherwise, a weight of 2 is assigned to an edge between two vertices in G* if the edge does not exist in G.

Traveling Salesman Problem

- This way, if we solve the complete graph G* for the Traveling Salesman problem, if there exists a Hamiltonian Circuit in the original graph G, the minimum weight tour in G* will involve only edges of weight 1. All such edges in the minimal weight tour of G* will exist in an Hamiltonian Circuit of G.

- The polynomial time reduction from a HCP to a TSP is involved in:
  - Given an weighted graph G for the Hamiltonian Circuit problem, the construction of the complete graph G* for the Traveling Salesman problem:
    - For every edge (u, v) in G, include (u, v) in G* and weight (u, v) = 1
    - For every edge (u, v) not in G, include (u, v) in G* and weight(u, v) = 2

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

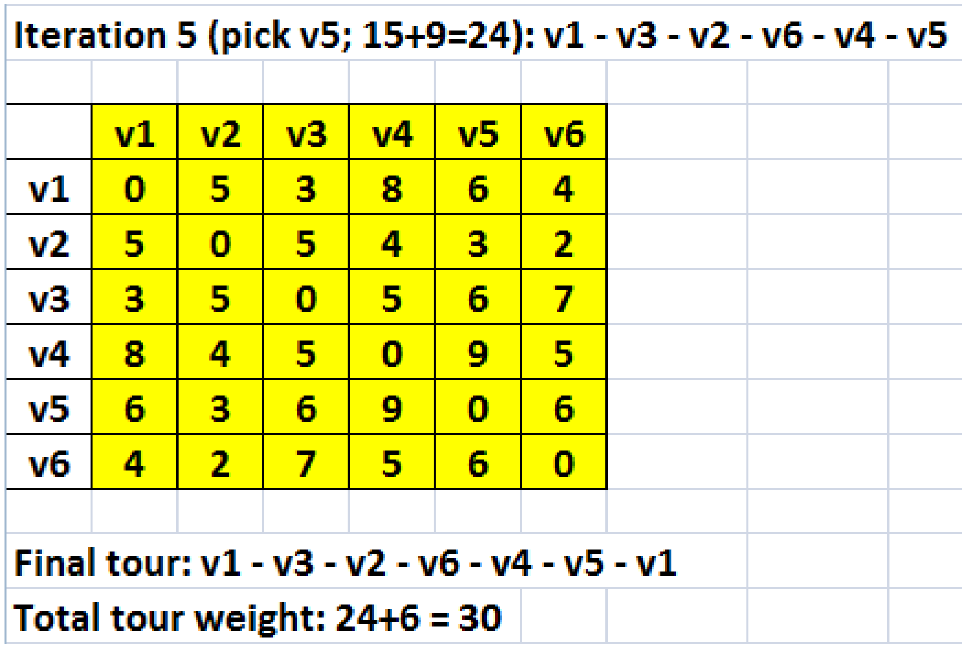
Iteration 3 (pick v6; 8+2=10): v1 - v3 - v2 - v6

	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

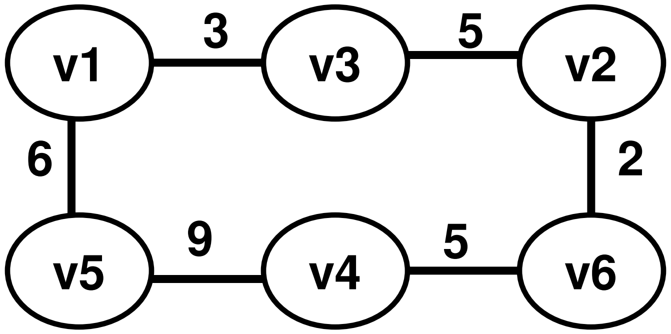
NN Heuristic Example (contd..)

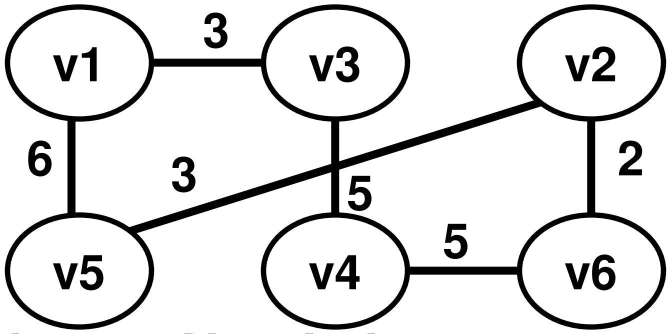

Iteration 5 (pick v5; 15+9=24): v1 - v3 - v2 - v6 - v4 - v5

	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

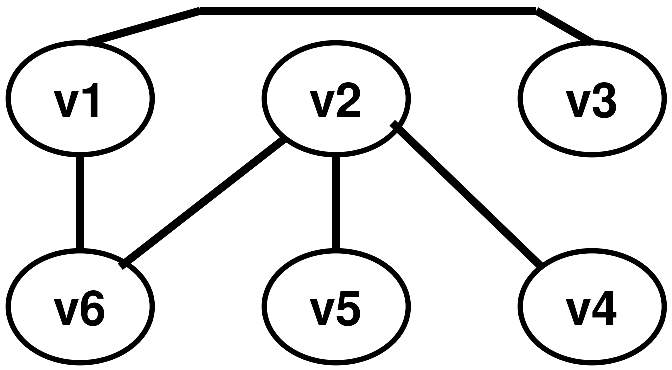
Heuristic # 2 for the TSP Problem Twice-around-the Tree Algorithm

- **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

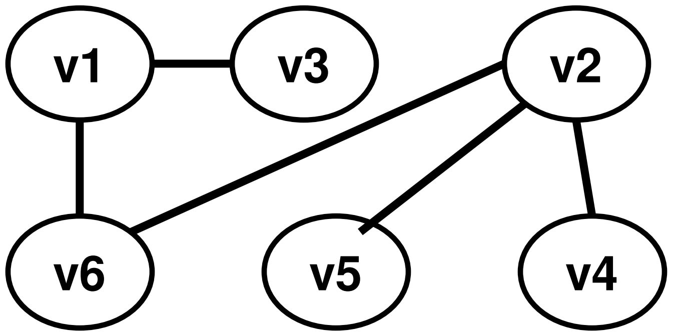
edge wt	edge		edge wt	edge
2	v2 - v6		5	v4 - v6
3	v1 - v3		6	v1 - v5
3	v2 - v5		6	v3 - v5
4	v1 - v6		6	v5 - v6
4	v2 - v4		7	v3 - v6
5	v1 - v2		8	v1 - v4
5	v2 - v3		9	v4 - v5
5	v3 - v4			

Step 1: MST of the Graph



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

MST (vertices rearranged) from Step 1

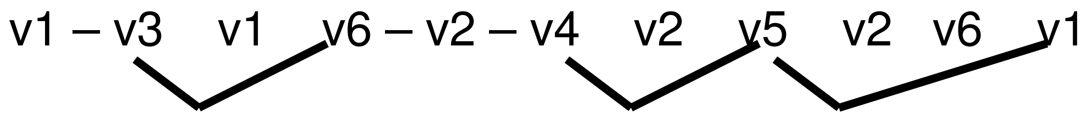


Step 3: Optimizing the DFS Walk

Tour from Step 2:

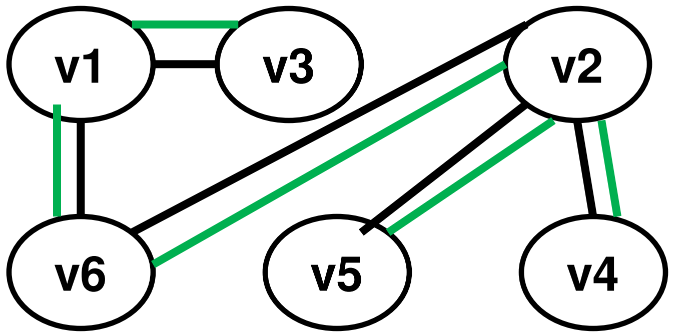
v1 – v3 – v1 – v6 – v2 – v4 – v2 – v5 – v2 – v6 – v1

Optimized Tour:



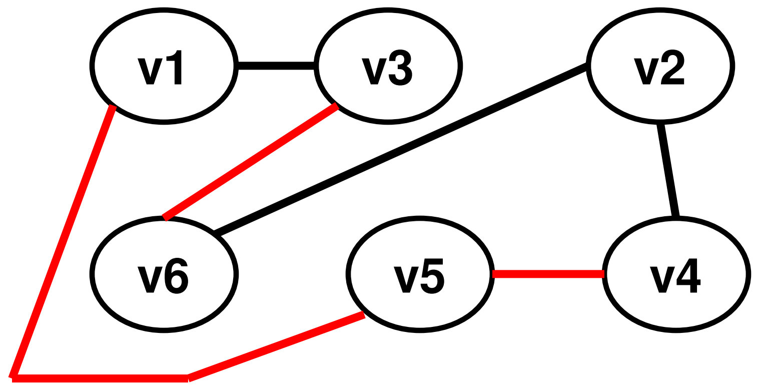
Step 2: DFS Traversal of the MST

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



v1 – v3 – v6 – v2 – v4 – v5 – v1

Tour Weight: 31



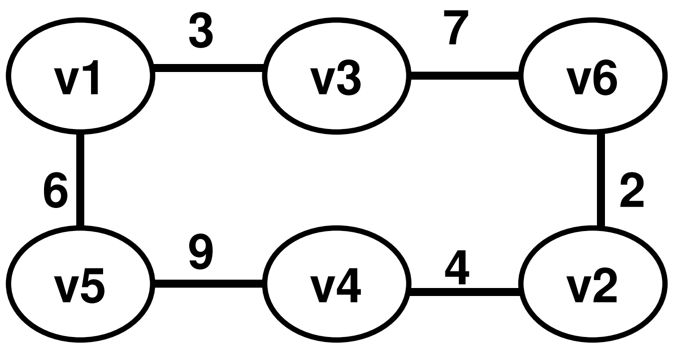
Heuristic # 2 for the TSP Problem Twice-around-the Tree Algorithm

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			

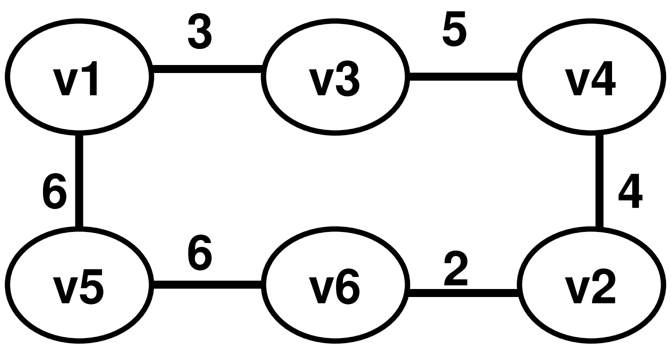
v1 – v3 – v6 – v2 – v4 – v5 – v1

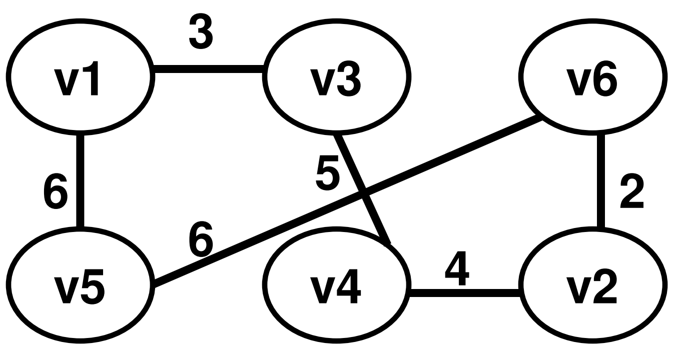
Tour Weight: 31



Improved Tour Weight: 26

Improvement using 2-Change





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

- Let w(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.

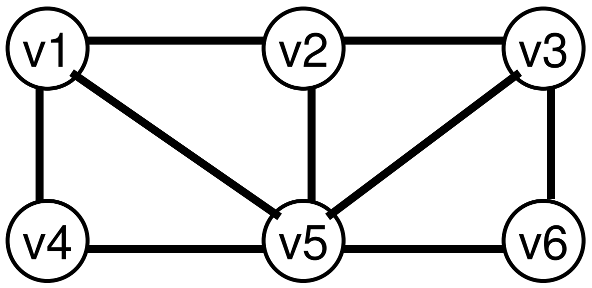
Independent Set, Vertex Cover, Clique

- 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.

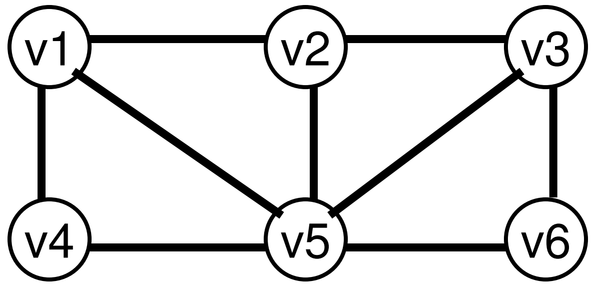
Approach to find the Independent Set, Vertex Cover and Clique for a Graph G

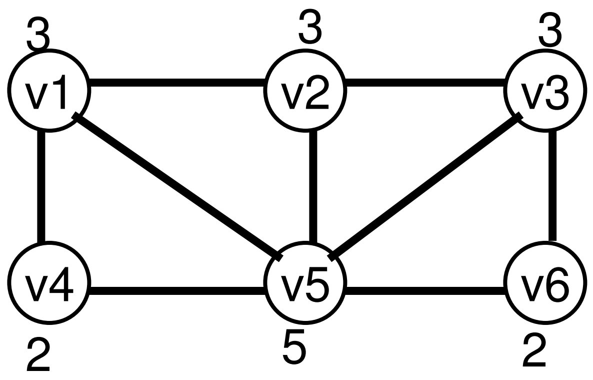
1. Find the Independent Set, IS, of graph G using the Minimum Neighbors Heuristic
2. The Vertex Cover of G is V – IS, where V is the set of vertices in G
3. Find the Complement Graph G* of G and run the Minimum Neighbors Heuristic on it. The Independent Set of G* is the Clique of G.

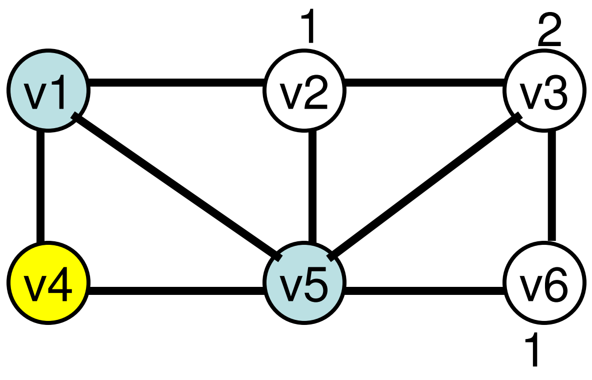
Proof for the Polynomial Reductions

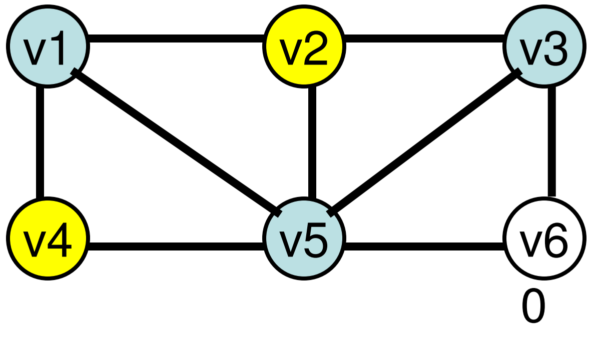
- **Clique $\leq_P$ Independent Set**
- Let G* be the complement graph of G. There exists an edge in G* if and only if there is no edge in G.
- Determine a Maximal Independent Set C* of G*. There exists no edge between any two vertices in C* of G*. ==> There exists an edge between the two vertices of 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.

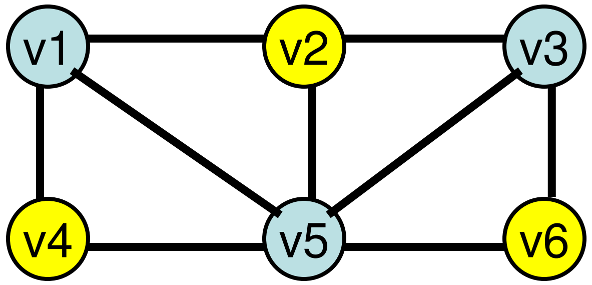
Example 1 to Find Independent Set using the Minimum Neighbors Heuristic











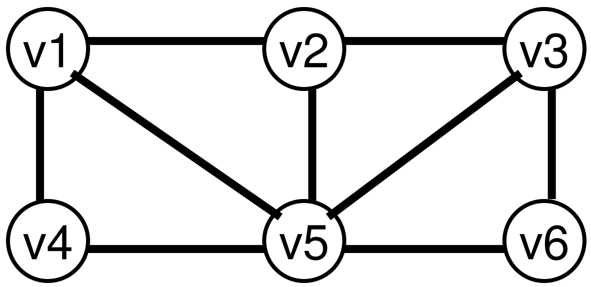
<u>Idea:</u> Give preference to vertices with minimal number of (uncovered) neighbors to be part of the Independent Set. A vertex is said to be covered if itself or any of its neighbors in the Independent Set.

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

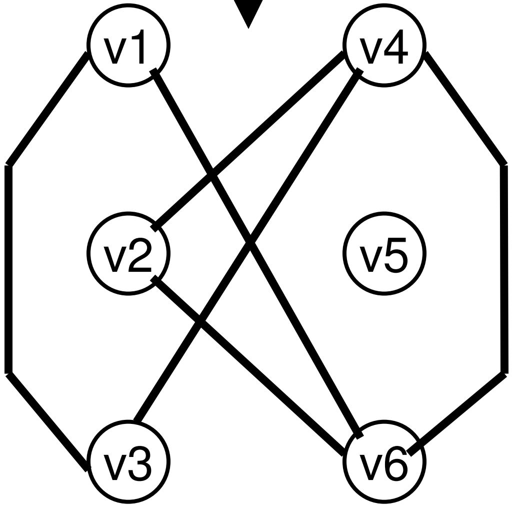
This is also the Maximal Independent Set (i.e., there exists no Independent Set of size 4 or more for the above graph). However, the heuristic is not guaranteed in general to give a maximal Independent set.

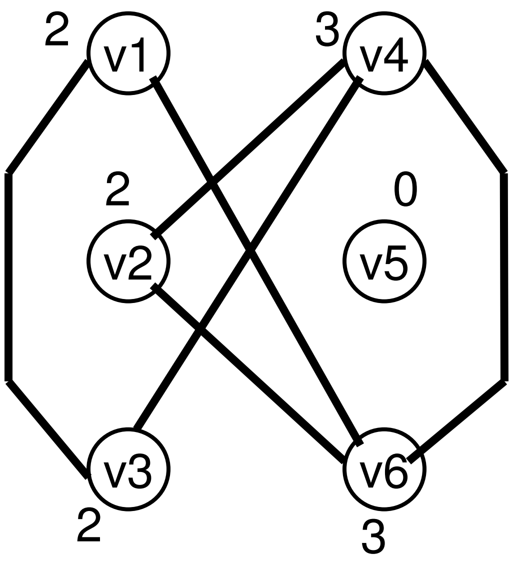
Vertex Cover = {v1, v3, v5}

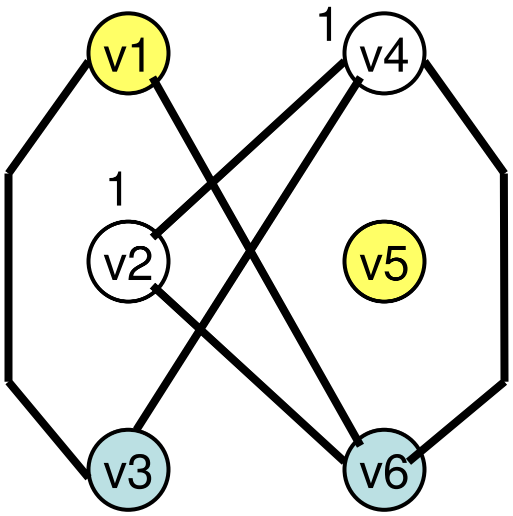
Given G ------->



Find G*, complement of G

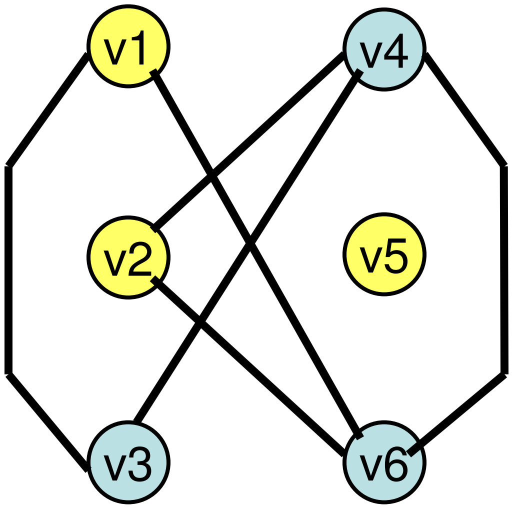




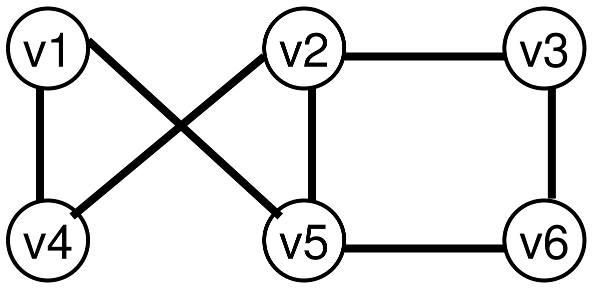


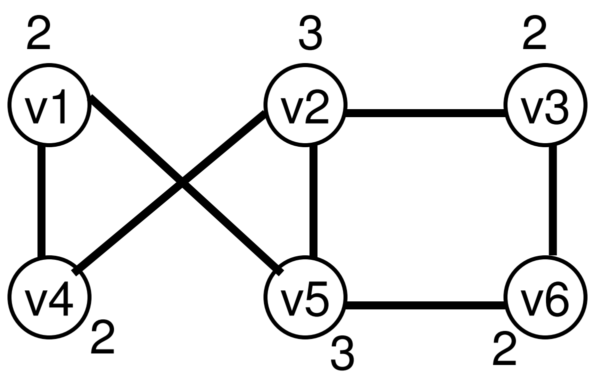
Example 1 to Determine a Clique Using the Minimum Neighbors Heuristic to Approximate an Independent Set

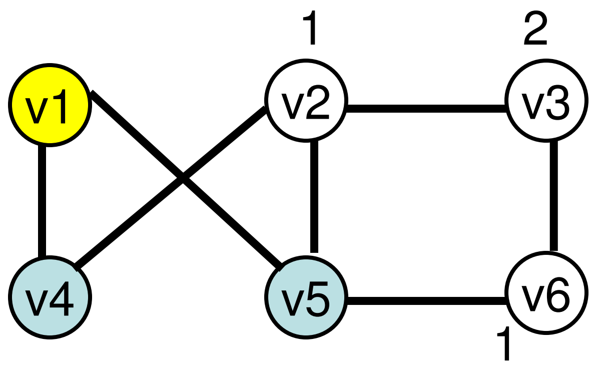
{v1, v2, v5} is an Independent Set in G* and it is a clique in G.

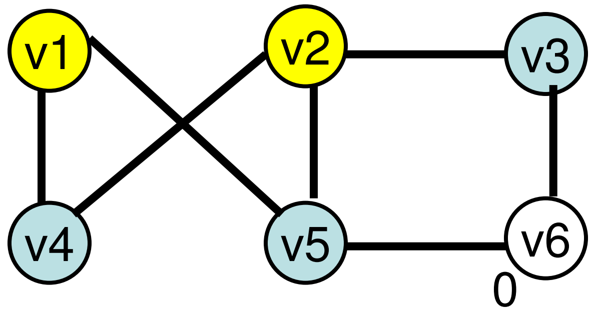


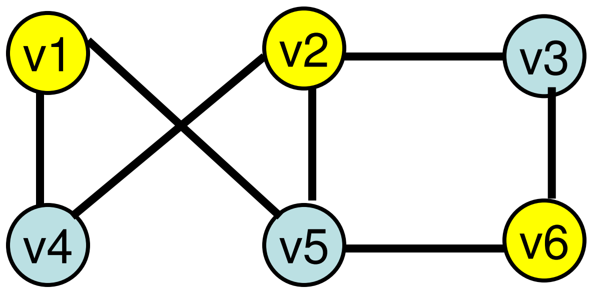
Example 2 to Find Independent Set using the Minimum Neighbors Heuristic







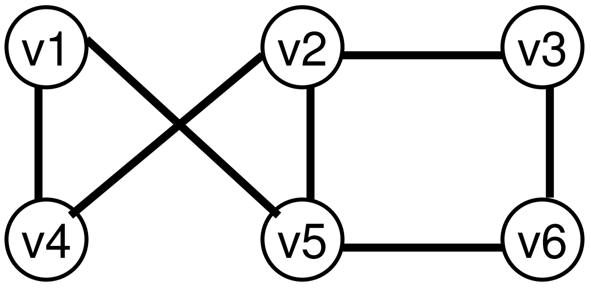




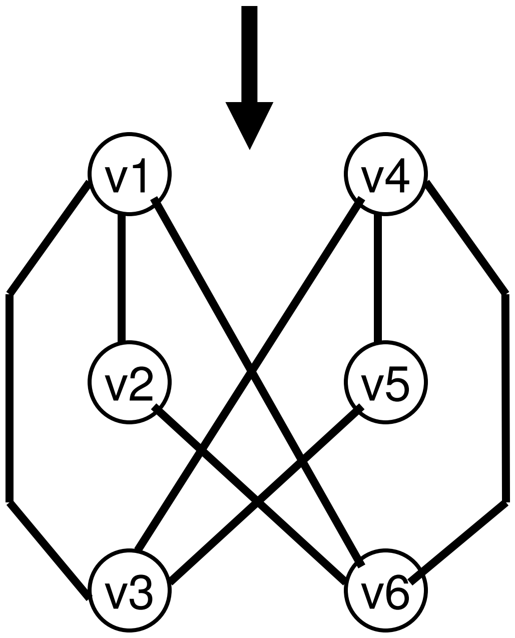
Independent Set = {v1, v2, v6}

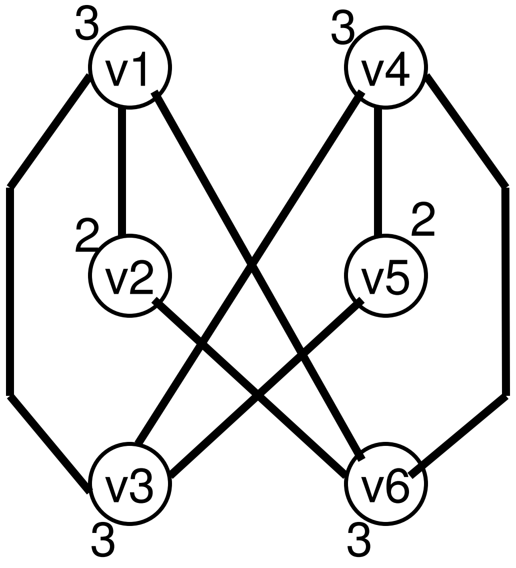
Vertex Cover = {v3, v4, v5}

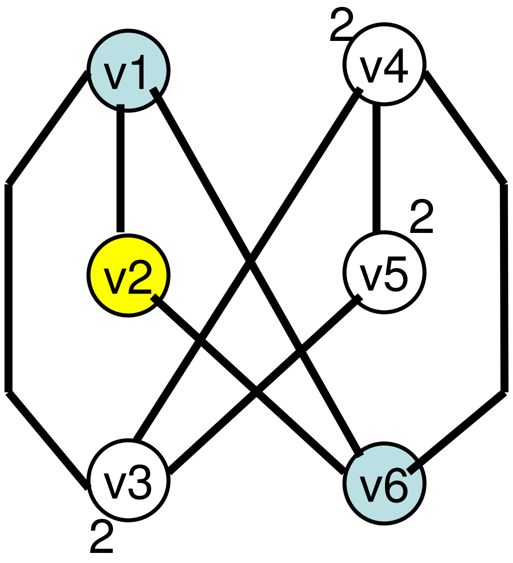
Given G ------->

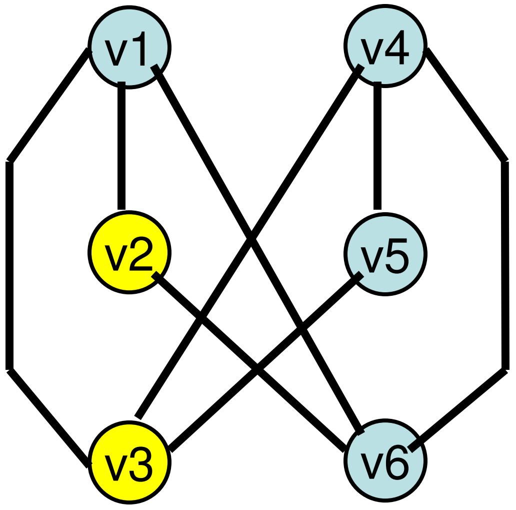


Find G*, complement of G







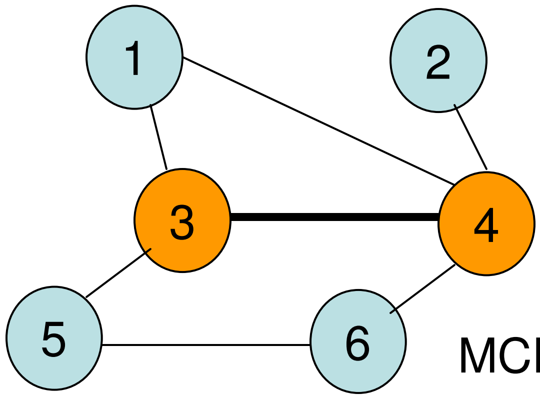


Independent Set of G* = {v2, v3}
Clique of G = {v2, v3}

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 like that of the unit disk graph is NP-complete.



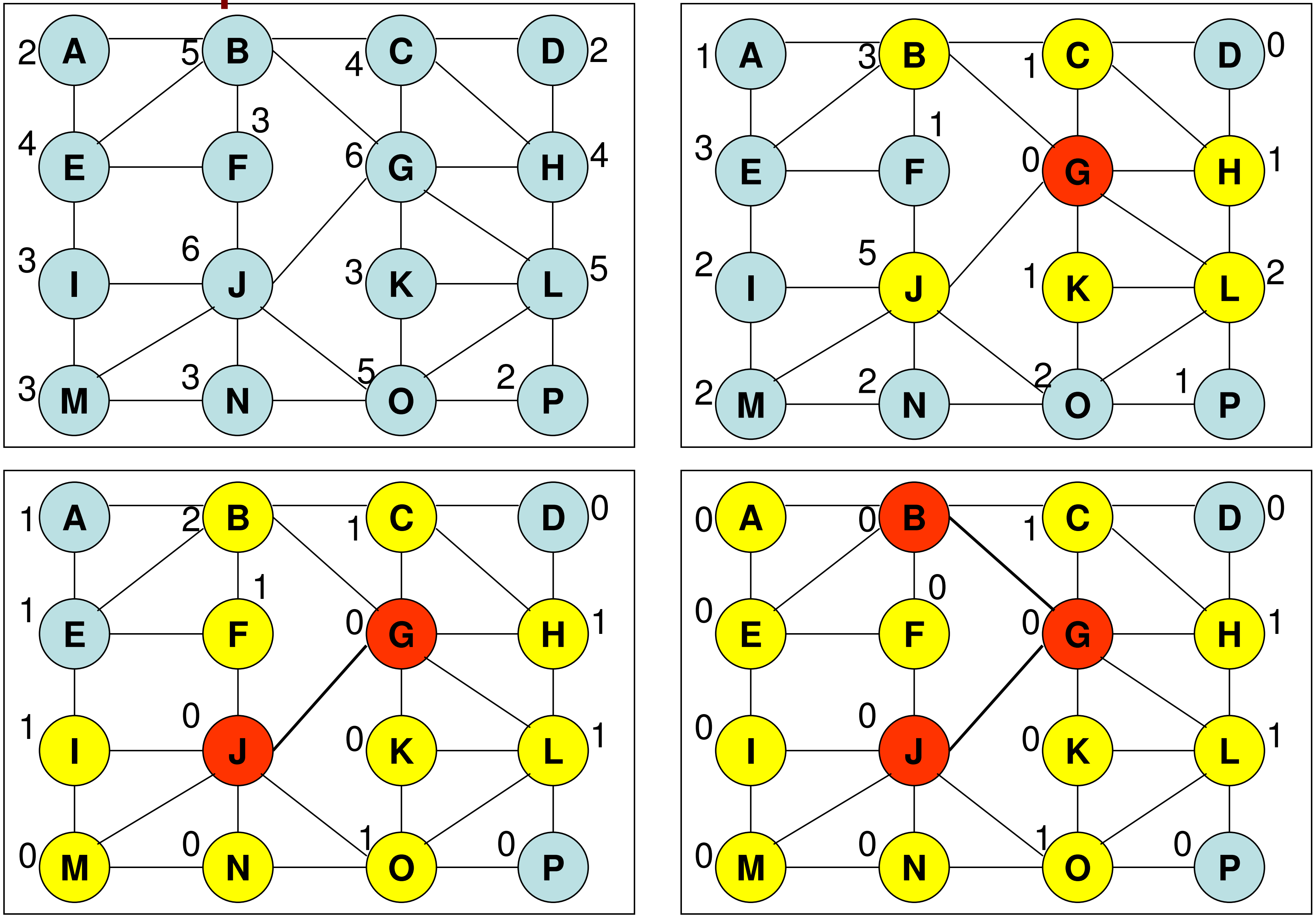
MCDS = [3, 4]

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

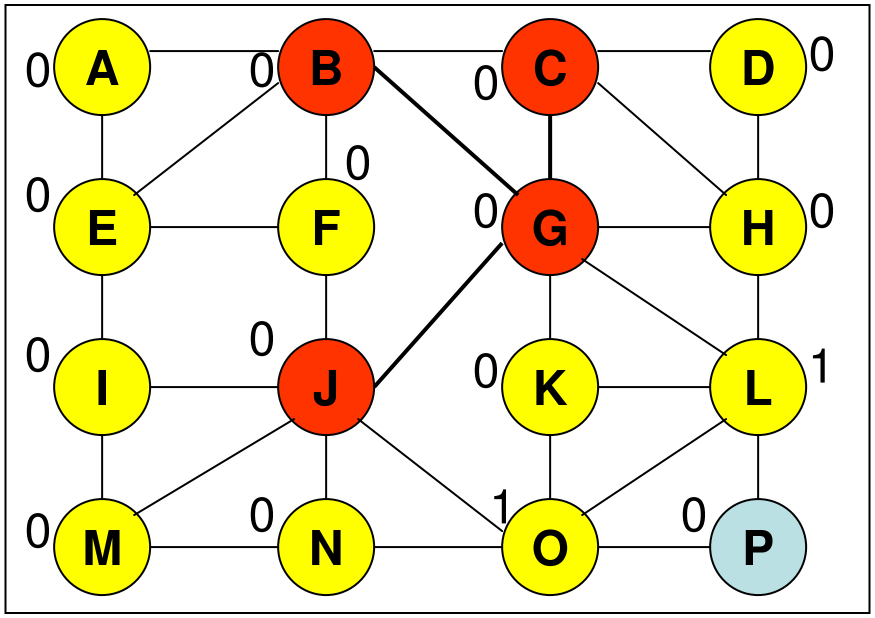
Heuristic to Approximate a MCDS

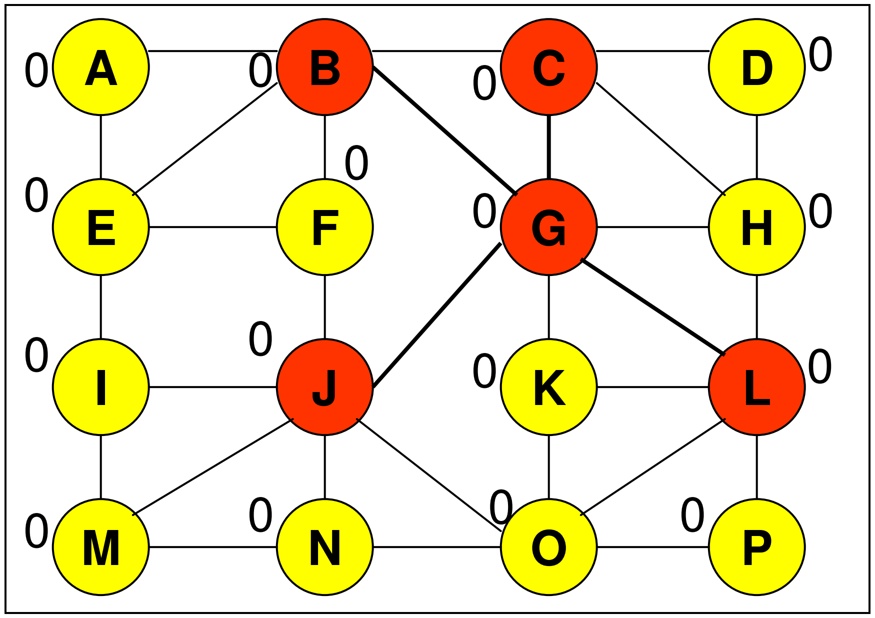
```
Input: Graph $G = (V, E)$, where $V$ is the vertex set and $E$ is the edge set.
       Source – vertex, $s$  $V$.
Auxiliary Variables and Functions:
       CDS-list, Covered-list, Neighbors($v$) for every $v$ in $V$.
Output: CDS-list
Initialization: Covered-list = {$s$}, CDS-list = $\Phi$
Begin d-MCDS
  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
```

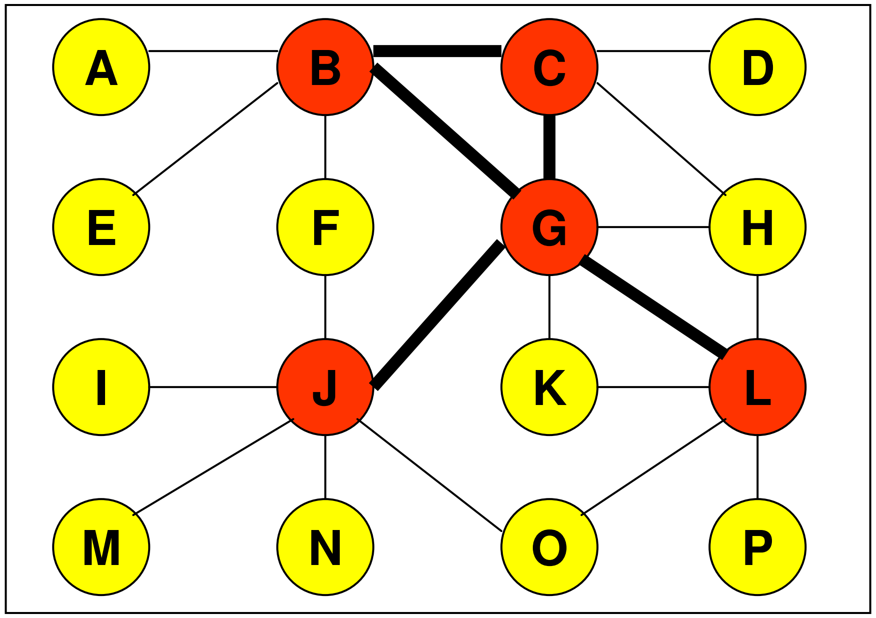
Example for *d-MCDS* Heuristic



Example for *d-MCDS* Heuristic







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