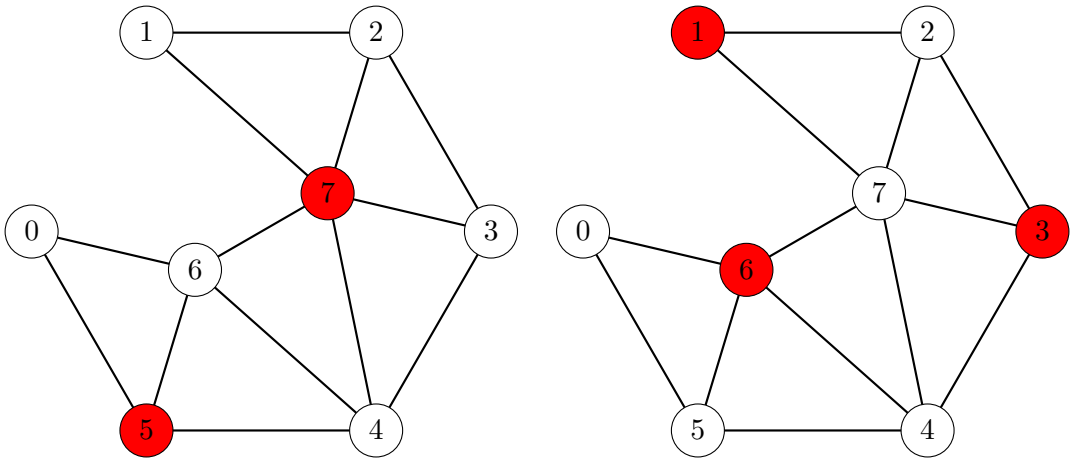
CSC 226 FALL 2015
ALGORITHMS AND DATA STRUCTURES II
ASSIGNMENT 5 - PROGRAMMING
UNIVERSITY OF VICTORIA

1 Programming Assignment

The assignment is to implement an algorithm to find an independent set of maximum size in an undirected graph. A Java template has been provided containing an empty method `MaximumIndependentSet`, which takes a single argument consisting of an adjacency matrix for an undirected graph G. The expected behavior of the method is as follows:

Input: A $n \times n$ array G representing a graph.
Output: An independent set S of maximum size with respect to G.

An *independent set* S of a graph G is a set of vertices of G such that no two vertices in S are adjacent. The diagrams below show different independent sets of a sample graph.



The largest independent set of the graph above has size 3, so the independent set on the right is maximum. The problem of finding a maximum independent set for an arbitrary graph is NP-hard, and no polynomial time algorithms are known to solve it.

You must use the provided Java template as the basis of your submission, and put your implmentation inside the `MaximumIndependentSet` method in the template. You may not change the name, return type or parameters of the `MaximumIndependentSet` method. You may add additional methods as needed. The `main` method in the template contains code to help you test your implementation by entering test data or reading it from a file. You may modify the `main` method to help with testing, but only the contents of the `MaximumIndependentSet` method (and any methods you have added) will be marked, since the `main` function will be deleted before marking begins. Please read through the comments in the template file before starting.

2 Input Format

The input format used by the testing code in `main` consists of the number of vertices n followed by the $n \times n$ adjacency matrix. The graph in the examples in the previous section would be specified as follows:

```
8
 0   0   0   0   0   1   1   0
 0   0   1   0   0   0   0   1
 0   1   0   1   0   0   0   1
 0   0   1   0   1   0   0   1
 0   0   0   1   0   1   1   1
 1   0   0   0   1   0   1   0
 1   0   0   0   1   1   0   1
 0   1   1   1   1   0   1   0
```

3 Vertex Sets

The return type of the `MaximumIndependentSet` function is the provided class `VertexSet`. A `VertexSet` object represents a set of vertices of the input graph G, and the `VertexSet` class contains methods to add vertices, remove vertices and copy the set. You may not modify any aspect of the provided `VertexSet` class, and your implementation must return a `VertexSet` instance. However, you are not required to use `VertexSet` anywhere else in your implementation (and you may find that alternative data structures are more efficient).

4 Basic Algorithm

The simplest algorithm to find a maximum independent set of a graph G on n vertices generates all subsets S of the vertices of G and tests whether each subset is an independent set, then returns the largest such set found. Since there are 2^n possible subsets of the n vertices of G, the naive algorithm becomes infeasible for relatively small values of n.

Although no polynomial time algorithm is known for the independent set problem, it is possible to improve on the brute force algorithm above. The resulting algorithm will still be exponential, but have better performance in practice. One simple improvement is to use the definition of independent set to guide the selection of a set of vertices. The pseudocode below gives a very basic algorithm to find the maximum independent set with a recursive search. For each vertex v_i, the algorithm alternately tries adding v_i to the independent set (if possible) and leaving v_i out of the independent set. For each choice of v_i's status, the recursion continues to the next level to decide the status of v_{i+1}. When recursion reaches level n, the status of every vertex has been decided, and if the resulting independent set is smaller than the best set found so far, it becomes the new best set.

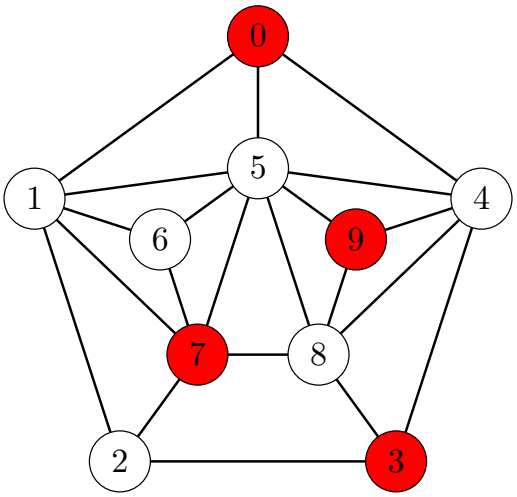
```
1:  procedure FindSets(G, B, S, v)
2:      n ← |V(G)|
3:      {If v = n, then all vertices have been processed, so the set S is complete.}
4:      if v = n then
5:          {If S is larger than the current best set, set the new best set to be a copy of S.}
6:          if |S| > |B| then
7:              Set B to be a copy of S.
8:          end if
9:          return
10:     end if
11:     {Case where v is not added to S}
12:     {Recurse to vertex v + 1}
```

```
    FindSets(G, B, S, v + 1)
    {The vertex v can be added to S if none of its neighbours are already in S.}
    if none of the neighbours of v are in S then
        {Case where v is added to S}
        Add v to S.
        {Recurse to vertex v + 1}
        FindSets(G, B, S, v + 1)
        Delete v from S.
    end if
end procedure
procedure MaximumIndependentSet(G)
    {Create a set S to represent the current set}
    S ← Empty Set
    {Create a set B to store the best set found so far}
    B ← Empty Set
    {Start the recursive algorithm on vertex 0}
    FindSets(G, B, S, 0)
    {Return the best set found}
    return B
end procedure
```

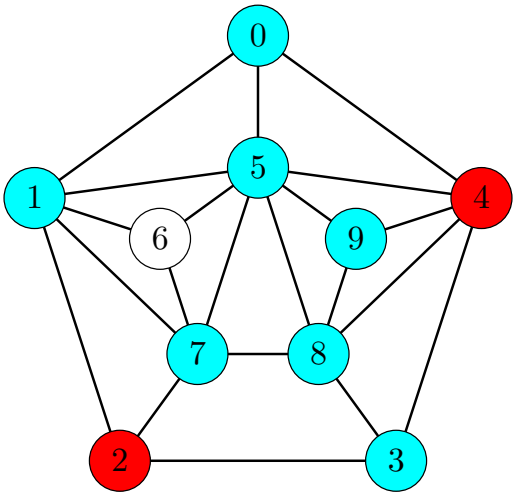
5 Intermediate Algorithm

The basic algorithm performs better than the brute force approach by only considering independent sets (since the recursive process only adds a vertex v_i to the set when it is eligible). However, the basic algorithm does not take advantage of some information discovered by the search process.

For example, if an independent set of size k is found, then the algorithm can end a branch of recursion early if it becomes clear that no set of size larger than k will ever be found there. Consider the graph below.



The highlighted independent set is maximum, but the algorithm will continue searching for a larger set after finding the one above (since without a full search, there is no way of knowing whether the maximum has been found). Eventually, the algorithm will reach the configuration shown below.



The set S (shown in red) contains two vertices: v_2 and v_4. The basic algorithm would try to build all possible independent sets which include vertices v_2 and v_4. However, none of the vertices adjacent to v_2 and v_4 (shown in light blue) would be eligible for an independent set. There is only one eligible vertex (vertex 6), so it is impossible to build an independent set larger than the best known size of 4. To save search time, the algorithm can terminate this branch of recursion early (and ignore all of the possible ways to extend the set $\{v_2, v_4\}$). This operation is called 'backtracking', and can dramatically increase the speed of recursive searches.

Modifying the basic algorithm to allow backtracking requires keeping track of which vertices are ineligible to be added to an independent set. One method of accomplishing this is creating a set F of 'forbidden' vertices. When a vertex u becomes adjacent to a vertex in S, it will be added to F, and no vertex in F can ever be added to S. The value $n - |F|$ gives the maximum number of eligible vertices remaining, so any independent set generated by the algorithm will have size at most

$$|S| + (n - |F|)$$

and therefore, if the algorithm reaches a point in recursion where

$$|S| + (n - |F|) \leq |B|$$

(where B is the best independent set found so far), then the algorithm can backtrack, since it is impossible to expand S into a larger independent set than B.

Pseudocode for the improved backtracking algorithm is given below.

```
1:  procedure FindSets(G, B, S, F, v)
2:      n ← |V(G)|
3:      {If v = n, then all vertices have been processed, so the set S is complete.}
4:      if v = n then
5:          {If S is larger than the current best set, set the new best set to be a copy of S.}
6:          if |S| > |B| then
7:              Set B to be a copy of S.
8:          end if
9:          return
10:     end if
11:     if |S| + (n − |F|) ≤ |B| then
12:         {S cannot be expanded into a larger independent set than B}
13:         return
```

```
    end if
    F' ← Copy of F
    Add v to F'
    {Case where v is not added to S}
    {Recurse to vertex v + 1}
    FindSets(G, B, S, F', v + 1)
    {The vertex v can be added to S if it is not in F}
    if v ∉ F then
        {Case where v is added to S}
        Add v to S.
        for each neighbour u of v do
            Add u to F'
        end for
        {Recurse to vertex v + 1}
        FindSets(G, B, S, F', v + 1)
        Delete v from S.
    end if
end procedure
procedure MaximumIndependentSet(G)
    {Create a set S to represent the current set}
    S ← Empty Set
    {Create a set B to store the best set found so far}
    B ← Empty Set
    {Create a set F to track forbidden vertices}
    F ← Empty Set
    {Start the recursive algorithm on vertex 0}
    FindSets(G, B, S, F, 0)
    {Return the best set found}
    return B
end procedure
```

6 Further Improvements

It is unlikely that this assignment will result in a polynomial time algorithm to find maximum independent sets for arbitrary graphs being discovered. However, there are many areas in which the intermediate algorithm described above could be improved. Although the resulting algorithm would still be exponential, a faster exponential algorithm would allow larger graphs to be processed.

To get full marks on this assignment, you must improve on the intermediate algorithm in some way. Any improvements to the algorithm are eligible, but your code must contain clear documentation of the improvements and how they differ from the original algorithm, in the form of comments in the following format:

```
/* IMPROVED: ... */
```

Simple improvements include changing the data structures used (for example, experimenting with an adjacency list representation) or altering the order in which vertices are processed. As presented in the pseudocode above, the algorithm performs a large number of redundant operations (such as copying sets), which could be eliminated to improve performance. To receive

all 15 marks in this section, you must improve the bounding or backtracking aspect of the algorithm. You may implement an idea from a published source (include a full citation in your comments). You are not permitted to change the signature (return type or parameters) of the `MaximumIndependentSet` function, so if you decide to use alternative data structures, you must perform the necessary conversions to make them work with the provided data types.

7 Test Datasets

Several collections of randomly generated graphs have been uploaded to conneX. Your assignment will be tested on graphs similar but not identical to the uploaded graphs. You are encouraged to create your own test inputs to ensure that your implementation functions correctly in all cases.

Some of the collections of graphs are too large or complicated for any of the algorithms described here to process in a reasonable amount of time (particularly the graphs with more than 100 vertices), but may be useful for measuring the performance of improvements you incorporate into your algorithm.

8 Sample Run

The output of a model solution on the graph above is given in the listing below. Console input is shown in blue.

```
Reading input values from stdin.
Reading graph 1
8
 0   0   0   0   0   1   1   0
 0   0   1   0   0   0   0   1
 0   1   0   1   0   0   0   1
 0   0   1   0   1   0   0   1
 0   0   0   1   0   1   1   1
 1   0   0   0   1   0   1   0
 1   0   0   0   1   1   0   1
 0   1   1   1   1   0   1   0
Graph  1: Maximum independent set found with size 3.
      S = 1 3 6
Processed 1 graph.
Total Time (seconds): 0.01
Average Time (seconds): 0.01
```

9 Evaluation Criteria

The programming assignment will be marked out of 50, based on a combination of automated testing and human inspection, following the criteria in the table below. To allow testing, the program must be able to compute the maximum independent sets of the 10 and 25 vertex collections of graphs posted to conneX in under 5 seconds per file, or it will not be possible to mark the program. Other than that requirement, there are no constraints on running time. To verify the accuracy of the submission, it will be tested with graphs which differ from the posted datasets.

Score (/50)	Description
$0 - 5$	Submission does not compile or does not conform to the provided template.
$6 - 20$	The implemented algorithm does not comply with the running time requirement above or is substantially inaccurate on the tested inputs.
$21 - 30$	The implemented algorithm is accurate on all tested inputs and uses the 'basic' algorithm outlined above.
$31 - 40$	The implemented algorithm is accurate on all tested inputs and uses the 'intermediate' algorithm outlined above (or an equivalent algorithm). If the algorithm differs from the intermediate algorithm above, it must use bounding criteria and backtracking.
$41 - 50$	The implemented algorithm improves substantially on the intermediate algorithm (and remains completely accurate on all tested inputs).

To be properly tested, every submission must compile correctly as submitted, and must be based on the provided template. You may only submit one source file. **If your submission does not compile for any reason (even trivial mistakes like typos), or was not based on the template, it will receive at most 5 out of 50**. The best way to make sure your submission is correct is to download it from conneX after submitting and test it. You are not permitted to revise your submission after the due date, and late submissions will not be accepted, so you should ensure that you have submitted the correct version of your code before the due date. conneX will allow you to change your submission before the due date if you notice a mistake. After submitting your assignment, conneX will automatically send you a confirmation email. If you do not receive such an email, your submission was not received. If you have problems with the submission process, send an email to the instructor **before** the due date.