1.Vertex Cover.

We use a greedy algorithm to solve this problem. We open an array 'vis' to record whether the current point is selected.

Set node 1 as the **root**, and start recursive search from the root.

Set x as the current point being searched. If all of x's child nodes are selected, then x can be deselected.

Otherwise, x must be selected and added to the set (leaf nodes do not need to be selected).

Here is the Pseudocode:

```
vis = new boolean[n]
// recursive function
Procedure dfs(x, son):
    bool allChildrenSelected = true;
    for each child in son[x]
        if not vis[child]: // if the child is not selected
            allChildrenSelected = false
            dfs(child, child.son) // recursively search the child
        end if
    end for
    if (not allChildrenSelected or son.size() == 0): // if x is a leaf node or not all children are selected
        vis[x] = true // select node x
    end if

for each x in tree:
    if vis[x] is true:
        print(x)
    end if
end for

dfs(1, 1.son)
```

Firstly, it the tree is a line, This algorithm is obviously correct.

Then we consider if we select a vertex that does not have to be selected, what will happen. The answer is we choose it's father is better. Because that, we can 'control' more edges, so that we can release more vertex.

2.Kernelization.

(a)

This rule is **safe**. Because hence Ψ does not contain the literal $\neg v_i$.

Even if v_i is set to false, it is not possible to satisfy any C_i

(b)

This rule is **safe**.

Because we can think of C/C' and C' as two separate parts. Then set $x = true, y = false, z = false$.

This will satisfy all c'.

(c)

This rule is **not safe**

If we set $x = false, y = true$ then $(x \vee y), (\neg x)$ will be both satisfied.

It is better than $(y \vee y), (\neg y)$.

3.Depth-bounded search trees 1.

We use a depth-bounded search tree algorithm. Let $k, G(V, E)$ be the parameters of this algorithm.

Each time, if $k \geq 0$ and we can't find any cycle of length four we return **true**. if k == 0 and we can find a cycle of length four we return **false**.

Then we chose a cycle of length four, delete the 4 points in this cycle in turn and go to the next level of recursion.

To find the cycle of length four, we can use dfs algorithm.

Here is the Pseudocode:

```
Procedure findCycle(G(V, E)):
    var startVertex;
    set ans;//Vertex set
    function dfs(u, length, set):
        if(length == 4):
            if(E(start, u) in G(V, E)):
                ans = set
                return true
            else:
                return false
            end if
        for each (u, v) in G(V, E):
            if(dfs(v, length + 1, set.insert(v))) return true
        end for
        return false

    for(each x in V):
```

```
        startVertex = x
        if(dfs(x, 1, x)) return ans
    return false

Procedure limDfs(k, G(V, E)):
    set = findCycle(G(V, E)
    if(set == false):
        return true
    if(k == 0):
        return false

    for each x in set:
        if(limDfs(k - 1, G(V, E).delete(x))):
            return true

    return false
```

I think the only point that needs to be explained is that we only need to find one cycle to traverse the points in. Since we need to delete the ring anyway, deleting it as soon as we find it is positive determination.

The time complexity of this algorithm is $O(n^2 * 4^k)$, find the cycle is n^2, the size of the search tree is 4^k, but it is the worst case and will not be reached in most cases.

4. Depth-bounded search trees 2.

We use a depth-bounded search tree algorithm. Let $k, T(V, E)$and H be the parameters of this algorithm.

Each time if $k \geq 0$ and we can't find a pair of $(s_i, t_i) \in H$ that s_i, t_i in the same tree, return **true**.

If $k = 0$ and we can find a pair of $(s_i, t_i) \in H$ that s_i, t_i in the same tree, return **false**.

Otherwise, we find a path s_i to t_i satisfy $(s_i, t_i) \in H$ and s_i, t_i in the same tree.

Then iterate through each edge on this path, delete them in turn and enter recursion, return true if there is a true, otherwise return false.

Here is the Pseudocode:

```
Procedure findPath(G(V, E), H):
    dfs each tree and give it a color/id
    for each (s, t) in H:
        if(color[s] = color[t]):
            return path(s, t)
        end if
    end for
    return false


Procedure limDfs(k, G(V, E), H):
    path = findPath(G(V, E), H)
    if(path == false):
```

```
        return true
    end if
    if(k == 0):
        return false
    end if
    for each e in path:
        if(limDfs(k - 1, G(V, E).delete(e), H)):
            return true
        end if
    end for

    return false
```

The time complexity of this algorithm is $O(n * n^k) = O(n^{k+1})$, because the path size may be n in the worst case.

Example:

Consider the under tree and $H = \{(1,6),(3,11),(4,9),(7,12),(10,13)\}, k = 3$

```
       1
     / | \
    2  3  4
      /|  | \
     5 11 9  8
    /|   \   |
   6 7   12  13
        / \
       10 14
```

In the ideal case, we find a path $(1-3-5-6)$, then we delete edge $(1-3)$, the forest will be:

```
     1
   /  \
  2     4
        | \
       9   8
           |
           13
```

```
    3
   /|
  5 11
 /| |
6 7 12
    /\
  10 14
```

Then we find $(3-11)$ and cut it

```
      1
    /   \
  2       4
          | \
        9     8
              |
              13
```

```
    3
   /
  5
 /|
6 7
```

```
    11
    |
    12
    /\
  10  14
```

then we find $(4-9)$ and cut it

```
      1
    /   \
  2       4
           \
            8
            |
            13
```

```
    3
   /
  5
 /|
6 7
```

```
    11
    |
    12
    /\
  10  14
```

```
9
```

Now $k=0$ and we can't find any path, return true.

the algorithm end.