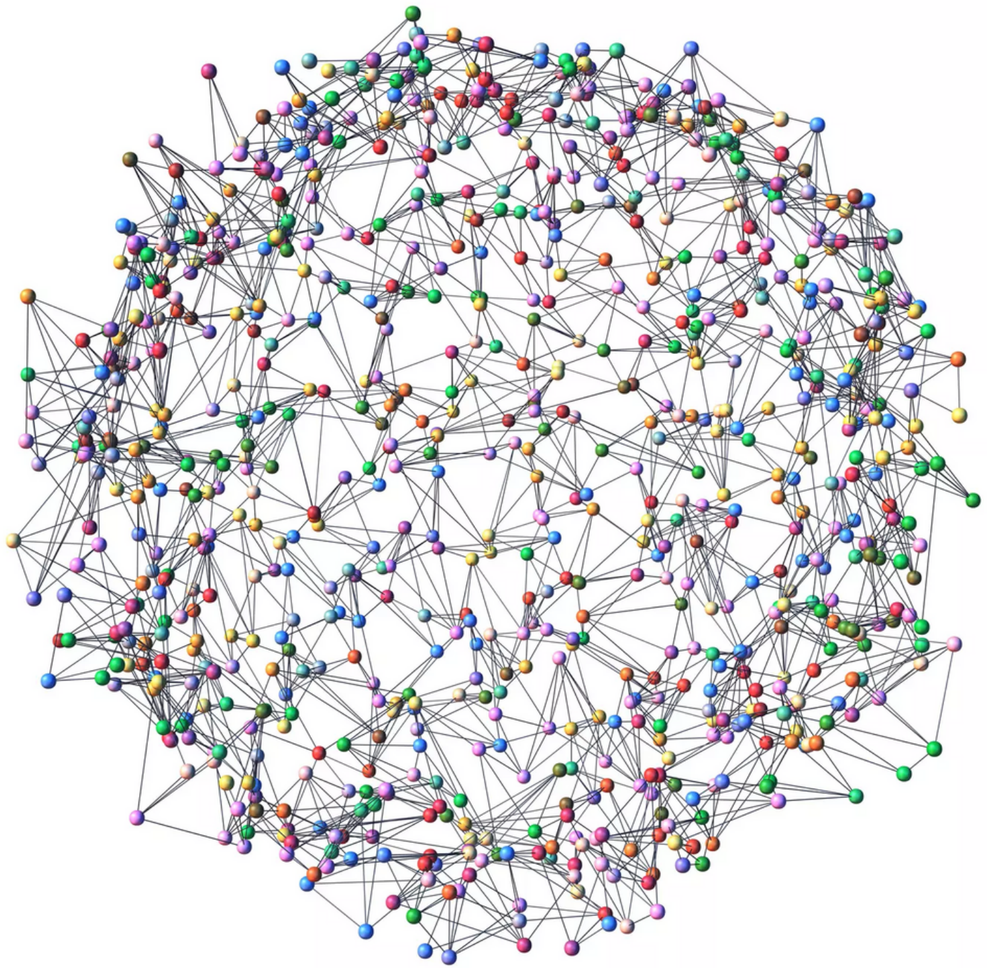
By: Dr. Meirav Zehavi

Topics in parameterized complexity



Rinat Vinokurov
Ofek Ben-David

Intoduction

Kernelization is an approach for pre-processing algorithms. It is usually achieved by replacing the algorithm's original input with a smaller one, called kernel.

Parameterized complexity is used to analyze the result of the kernelization algorithms.

Our project examines the **Vertex Cover** problem described in the "Parameterized Algorithms" textbook shown in class.

Vertex Cover

A vertex cover of a graph G is a set S of vertices of G such that every edge of G has at least one member of S as an endpoint.

The vertex cover problem is an NP-complete problem.

Our project

As mentioned in the introduction section, we have chosen to implement the vertex cover problem described in the "Parameterized Algorithms" textbook shown in class.

We used several random graphs as input and ran the following combinations on them:

		Recursive	Recursive with improvement	LP
Comparison 1	Without kernelization	✓	✓	
Comparison 2		✓	✓	✓
Comparison 3	With kernelization	✓	✓	✓

All the implementations mentioned were written in python.

The simple kernelization algorithm

Reduction VC.1. If G contains an isolated vertex v, delete v from G. The new instance is $(G - v, k)$.

Reduction VC.2. If there is a vertex v of degree at least $k + 1$, then delete v (and its incident edges) from G and decrement the parameter k by 1. The new instance is $(G - v, k - 1)$.

Reduction VC.3. Let (G, k) be an input instance such that Reductions **VC.1** and **VC.2** are not applicable to (G, k). If $k < 0$ and G has more than $k^2 + k$ vertices, or more than k^2 edges, then conclude that we are dealing with a no-instance.

The algorithm described above separates VC.1 and VC.2 into two different rules. Therefore, we iterate over all of the graph's vertices twice.

Our improvement of the simple kernelization algorithm

Even though VC.1 and VC.2 are indeed separate rules and each has a different purpose, they both iterate over the graph's nodes. Thus, we have decided to combine both of the rules under the same iteration over the graph's nodes.

> **Reduction VC.1.** If G contains an isolated vertex v, delete v from G. If there is a vertex v of degree at least $k + 1$, then delete v (and its incident edges) from G and decrement the parameter k by 1.
>
> **Reduction VC.2.** Let (G, k) be an input instance such that Reductions **VC.1** is not applicable to (G, k). If $k < 0$ and G has more than $k^2 + k$ vertices, or more than k^2 edges, then conclude that we are dealing with a no-instance.

In the original algorithm, the iteration over the graph's vertices for VC.1 is done before the iteration over the vertices for VC.2. In VC.1, we remove all the isolated vertices. Therefore the reduced graph after VC.1 doesn't affect VC.2, and we can combine both rules under the same iteration without compromising the algorithm's correctness.

The recursive algorithm

Given a graph G and an integer k.

1. Pick an arbitrary edge (v, u).
2. We branch, as follows, to find the OPT solution:
    a. Add v into the vertex cover:
        i. Decrease k by 1.
        ii. Delete v from the graph.
        iii. Run $rec\_VC(G', k')$.
    b. Add u into the vertex cover:
        i. Decrease k by 1.
        ii. Delete u from the graph.
        iii. Run $rec\_VC(G', k')$.

The correctness of this algorithm is based on the fact that for each edge (u,v) in G, either u is in the vertex cover or v is in the vertex cover.

Improvement of the recursive algorithm

Given a graph G and an integer k.

3. Pick $v \in V(G)$ with the maximum degree in G.
4. We branch, as follows, to find the OPT solution:
    a. Add v into the vertex cover:
        i. Decrease k by 1.
        ii. Delete v from the graph.
        iii. Run $rec\_opt\_VC(G', k')$.
    b. Add $N[v]$ into the vertex cover:
        i. Decrease k by $|N[v]|$.
        ii. Delete $N[v]$ from the graph.
        iii. Run $rec\_opt\_VC(G', k')$.

We can extend the observation from the simple recursive algorithm and say:
For a vertex v, any vertex cover of G must contain either v or all of its neighbors N(v).
Since we pick the vertex with the maximum degree, we guarantee to cover each iteration's maximum number of edges.

LPVC

For a graph G, we abtain the following linear programming instance:

$$
\begin{array}{lll}
\min & \sum_{v \in V(G)} x_v & \\
\text{subject to} & x_u + x_v \geq 1 & \text{for every } uv \in E(G), \\
& 0 \leq x_v \leq 1 & \text{for every } v \in V(G).
\end{array}
$$

The $LPVC(G)$ returns $\sum x_v$ of minimum value $vc^*(G)$, which leads to the vertex cover above LP algorithm described next.

Vertex cover above LP

Given a graph G and an integer k, we ask for a vertex cover of G of size at most k, but instead of seeking an FPT algorithm parameterized by k, the parameter now is k − vc∗(G).

1. Let w be the solution of $LPVC(G)$ and $vc^*(G)$ be the size of w.
2. If $vc^*(G) > k$, then:
    a. (G, k) is no-instance.
3. Reduction **VC.4.**
4. If w is all-1/2-solution, then:
    a. Reduction **VC.5.**
    b. Reduction **VC.4.**
5. Delete $V_0 \cup V_1$ from G and decrease k by $|V_1|$.
6. If w was all-1/2-solution, then:
    a. Reduction **3.8.**
7. Return $V_1 + VC\_above\_LP(G', k')$

- $(G', k') = the\ new\ instanse.$

LP - Reduction VC.4

Given a graph G and an optimum solution w. We define $x_v \in \{0, \frac{1}{2}, 1\}$ to be the value of v in w.

We divide G's vertices as follows:

1. $V_0 = \{v \in V(G) \,|x_v < \frac{1}{2}\}$
2. $V_{\frac{1}{2}} = \{v \in V(G) \,|x_v = \frac{1}{2}\}$
3. $V_1 = \{v \in V(G) \,|x_v > \frac{1}{2}\}$

LP - Reduction VC.5

Given a graph G and an integer k.

1. $OPT = \min_{v \in V(G)} \{LPVC(G - v)\}$
2. Let $v \in V(G)$ be the vertex for which OPT is the optimum solution.
3. Set the value $x_v = 1$.

LP - Reduction 3.8

Given a graph G.

1. Pick an arbitrary vertex v.
2. We branch, as follows, to find the OPT solution:
    a. Add v into the vertex cover:
        i. Decrease k by 1.
        ii. Delete v from the graph.
        iii. Run $VC\_above\_LP(G', k')$.
    b. Add $N[v]$ into the vertex cover:
        i. Decrease k by $|N[v]|$.
        ii. Delete $N[v]$ from the graph.
        iii. Run $VC\_above\_LP(G', k')$.

Randomized graphs as input

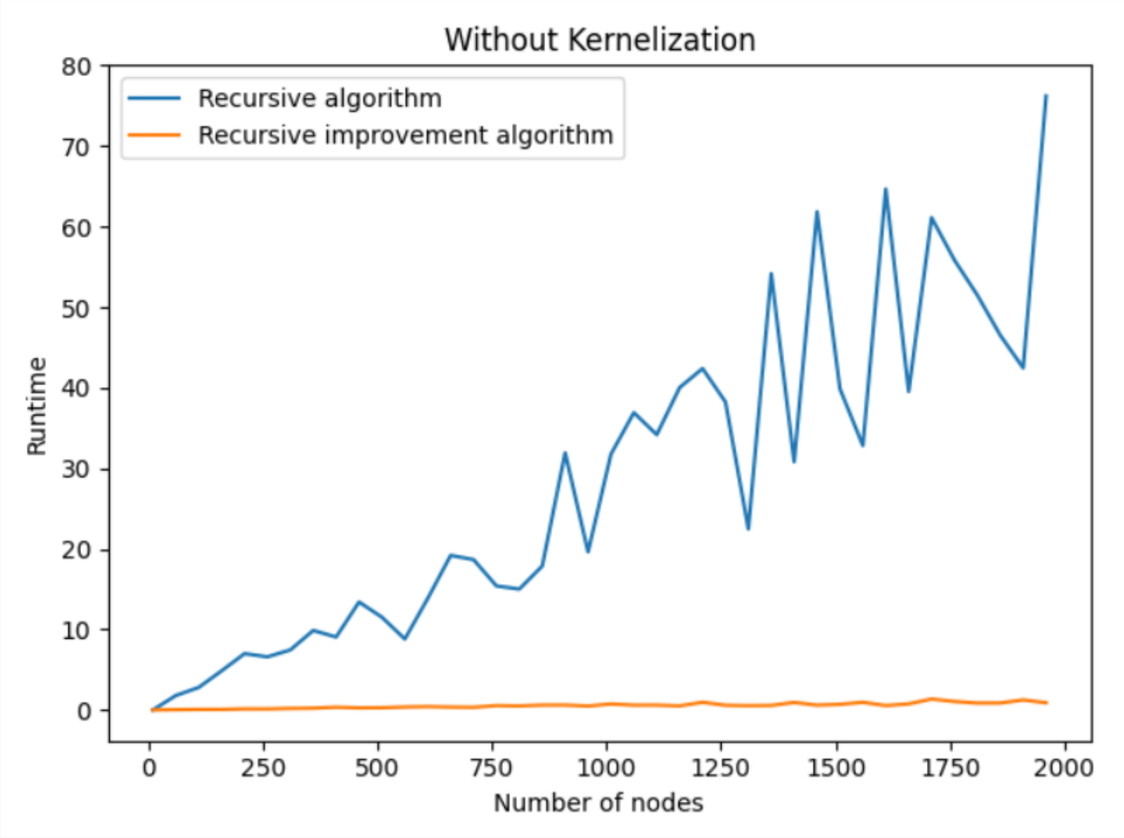
We ran each algorithm with k = 10 and a random graph, created as follows:

NODES: We started with a 10-nodes graph, added 10 more nodes to the next input graph, and so on until we reached a graph with 2000 nodes.

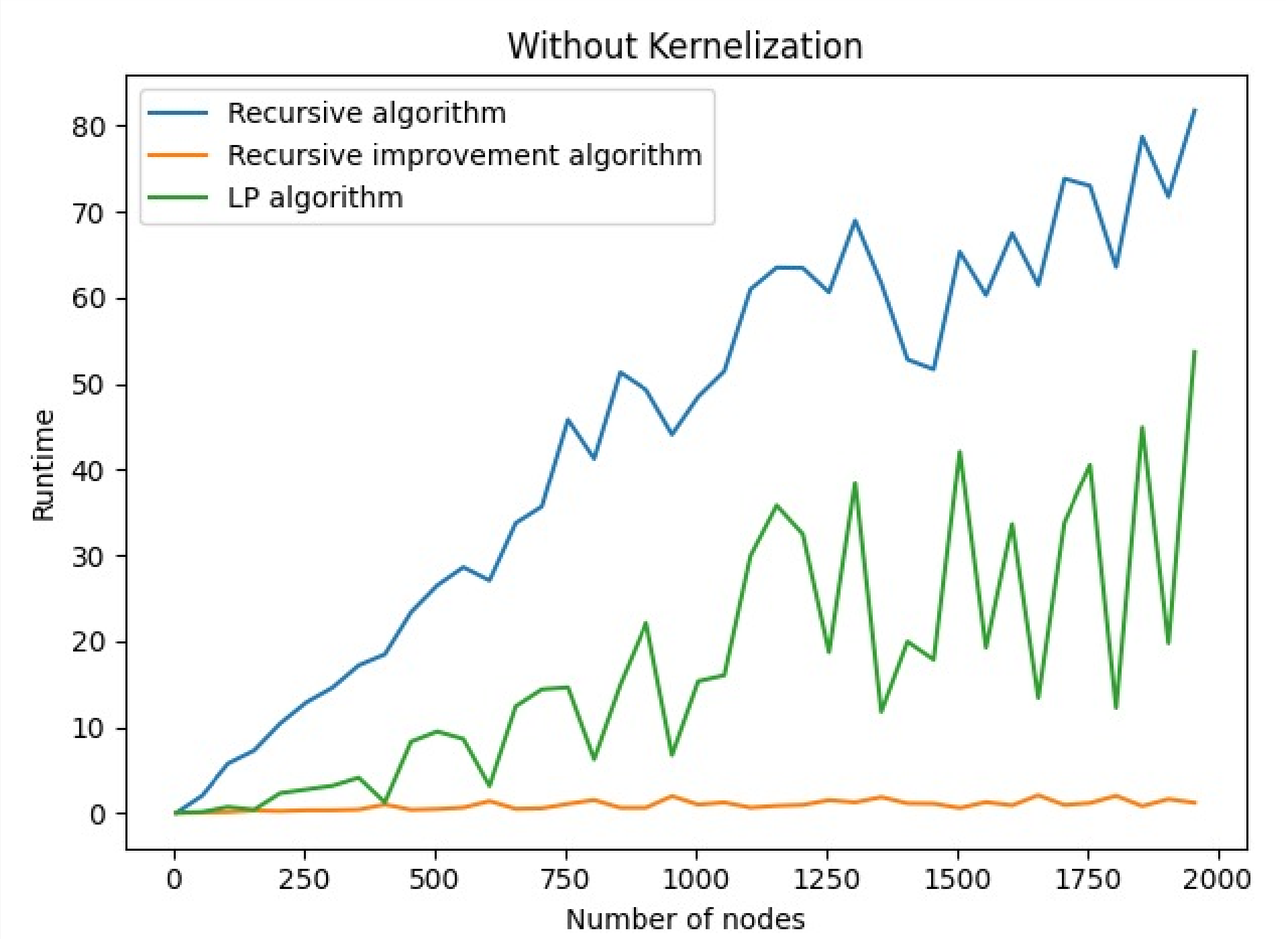
EDGES: Random number of edges between ((number of nodes) / 4) to (number of nodes).

```
k = 10
for n in range(10, 2000, 10):
    m = random.randint(int(n/4), n)
    G = nx.gnm_random_graph(n, m)
```

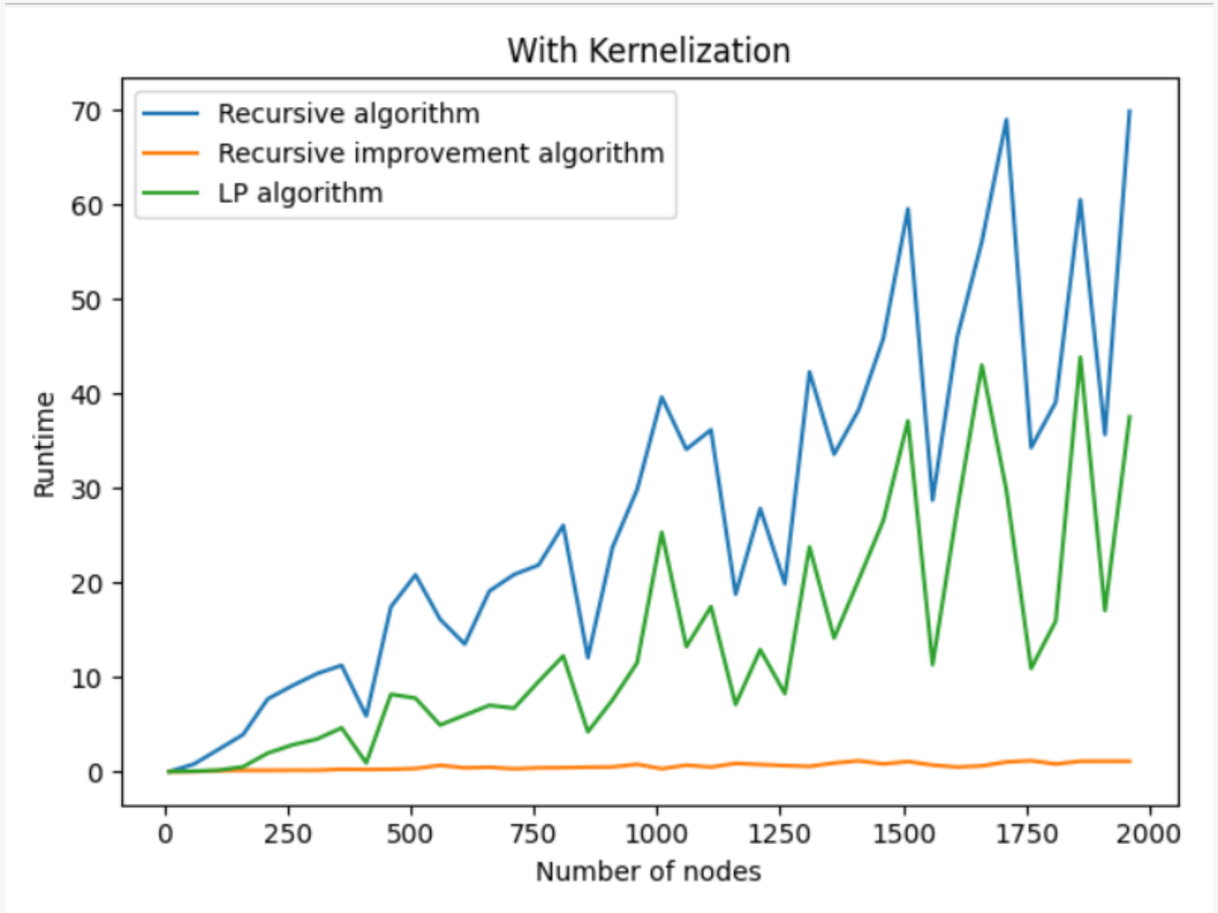
Recursive vs. Recursive with improvement WITHOUT Kernelization



Recursive vs. Recursive with improvement vs. LPVC WITHOUT Kernelization



Recursive vs. Recursive with improvement vs. LPVC WITH Kernelization



Kernelization

Random graphs with 100 nodes and 70 edges.
k = 10

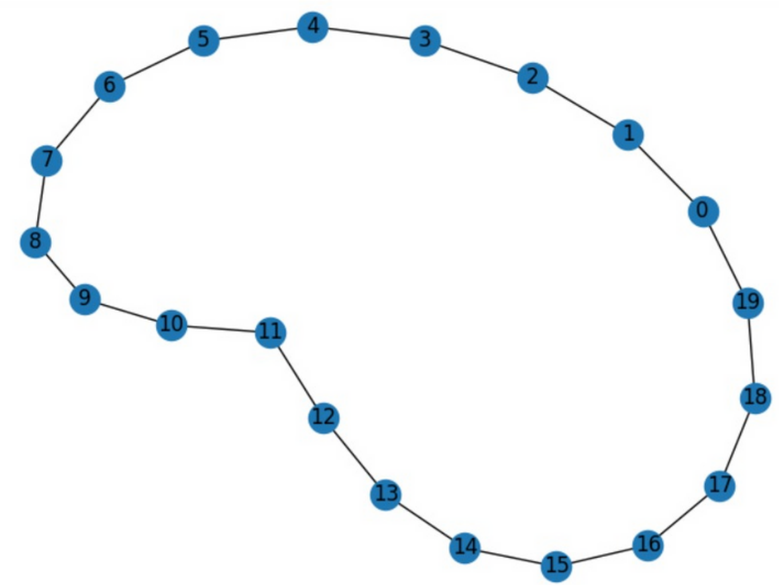
```
Recursive with opt without kernelization:
runtime: 225.1096355
result: [3, 71, 7, 43, 25, 9, 11, 68, 29, 23, 22, 0, 60, 34, 6, 35, 69, 39, 12, 45, 58, 10, 73, 80, 30, 86, 42]
Recursive with opt with kernelization:
runtime: 138.76215670000002
result: [3, 71, 7, 43, 25, 9, 11, 68, 29, 23, 22, 0, 60, 34, 6, 35, 69, 39, 12, 45, 58, 10, 73, 80, 30, 86, 42]
```

```
LP without kernelization:
runtime: 147.4080818
result: [5, 68, 51, 30, 3, 63, 9, 60, 56, 43, 8, 32, 0, 48, 80, 41, 17, 93, 26, 59, 29, 37, 24, 70, 42]
LP with kernelization:
runtime: 96.228241
result: [5, 68, 51, 30, 3, 63, 9, 60, 56, 43, 8, 32, 0, 48, 80, 41, 17, 93, 26, 59, 29, 37, 24, 70, 42]
```

Circle-graph as input

We ran each algorithm with k = 10 and a circle graph with 20 nodes.

The input graph



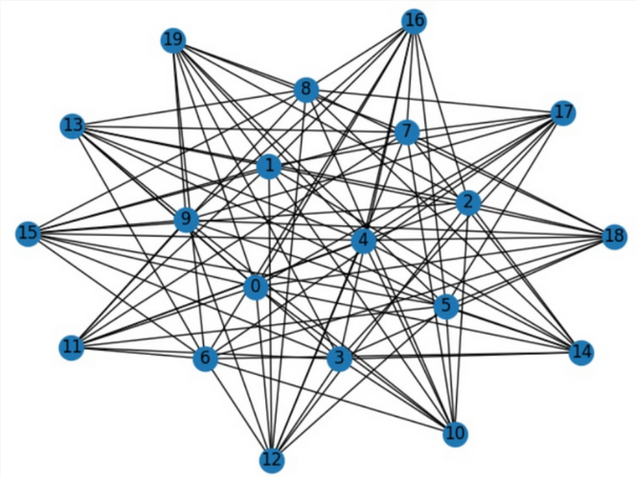
Our result

```
recursive without kernelization:
runtime: 0.6447075
result: [18, 16, 14, 12, 10, 8, 6, 4, 2, 0]
recursive opt without kernelization:
runtime: 0.08525129999999992
result: [17, 15, 13, 11, 9, 7, 5, 3, 1, 19]
LP without kernelization:
runtime: 0.043451600000000035
result: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

Complete-bipartite graph as input

We ran each algorithm with k = 10 and a complete bipartite graph with 10 nodes in each node set V1, V2.

The input graph



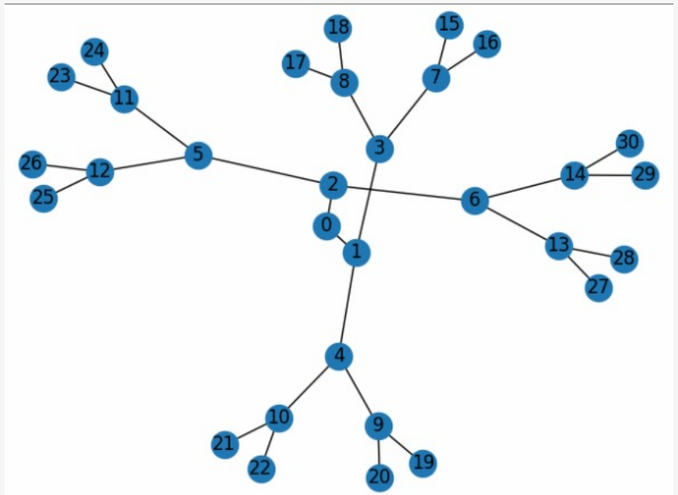
Our result

```
recursive without kernelization:
runtime: 1.4023086000000002
result: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
recursive opt without kernelization:
runtime: 0.014401599999999792
result: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
LP without kernelization:
runtime: 0.3098847
result: [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Balanced-tree graph as input

We ran each algorithm with k = 10 and a balanced tree graph with height = 4.

The input graph



Our result

```
recursive without kernelization:
runtime: 1.6887769
result: [14, 13, 12, 11, 10, 9, 8, 7, 2, 1]
recursive opt without kernelization:
runtime: 0.0899337
result: [14, 13, 12, 11, 10, 9, 8, 7, 2, 1]
LP without kernelization:
runtime: 0.1387573999999998
result: [1, 2, 7, 8, 9, 10, 11, 12, 13, 14]
```

Results on randomized graphs

First, we wanted to compare the recursive algorithm with the optimization with the recursive algorithm without optimization. As we have seen in the charts, we found a dramatic difference between the running times of the two algorithms.
In the optimized algorithm, we handled all of the neighbors in the same iteration instead of one by one, and as a result, we managed to reduce the runtime for each input.

Then, we compared the three parameterized algorithms, in order to determine which algorithm has the best runtime. We saw that for each input, the recursive algorithm with the optimization was better than the others, followed by the LPVC algorithm - with and without kernelization.
Since reduction 3.8 of the LPVC algorithm can be found in the recursive algorithm, and the LPVC contains additional reduction rules, meaning the LPVC executes more reduction steps than the recursive algorithm, it results in additional runtime.

In addition, we noticed by applying the kernelization prior to running the parameterized algorithms, we managed to improve the total runtime of the VC-solution algorithm. We achieved an improvement of ~ 40%.

Results on special types of graphs

Circle-graph

In a circle graph, the VC must be of at least n-1 nodes. We noticed that in such a graph, the recursive runtime without optimization was higher than the other two algorithms since we remove one edge in each iteration, resulting in n-iterations. However, the other two algorithms remove all of v's neighbors in each iteration.

Complete bipartite graph

In a complete bipartite graph, each node in one node group has an edge with all the other nodes in the other node group. Therefore, both in the optimized recursive algorithm and in the LPVC algorithm, regardless of the vertex we choose to examine in the current iteration, we can select all of its neighbors to the VC and return. Therefore we can observe that both the optimized algorithm and the LPVC have much better results than the non-optimized recursive algorithm.

Results on special types of graphs

Balanced-tree graph

Each node represents at most 3 edges (one connected to its parent and 2 for each child). In the non-optimized recursive algorithm, we handle one edge in each iteration. In contrast, in the other two algorithms, we remove all of the current vertex's edges - at most 3, or remove all of its neighbor's edges - at most 9. Therefore, we can dramatically reduce the runtime in the optimal solution since each iteration removes many more edges.