MAXIMUM DOMINATION OF K-VERTEX SUBSET IN TREES

Author: Yan-Tong Lin
Advisor: Prof. Chiu-Yuan Chen

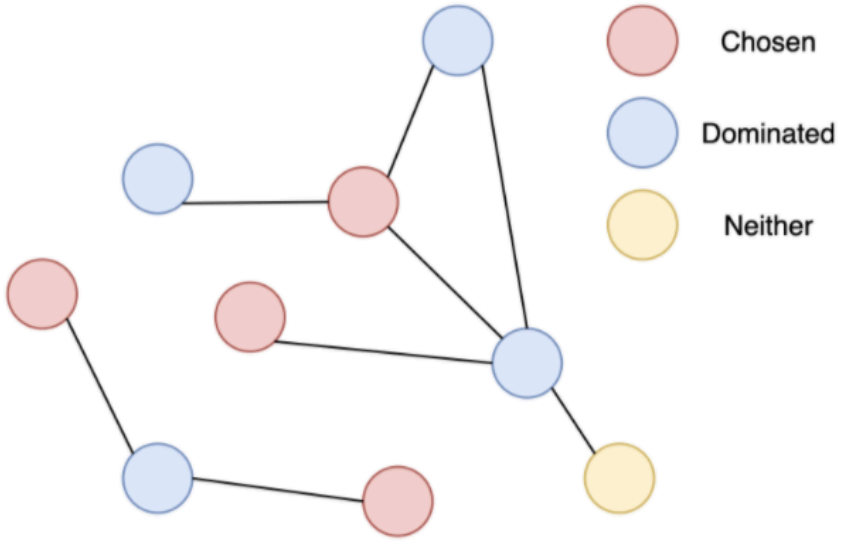
Overview

- Task Description
- Related Works
- Proposed Algorithm
  - *Vanilla DP*
  - *SMAWK's Optimization*
- Demos

THE TASK

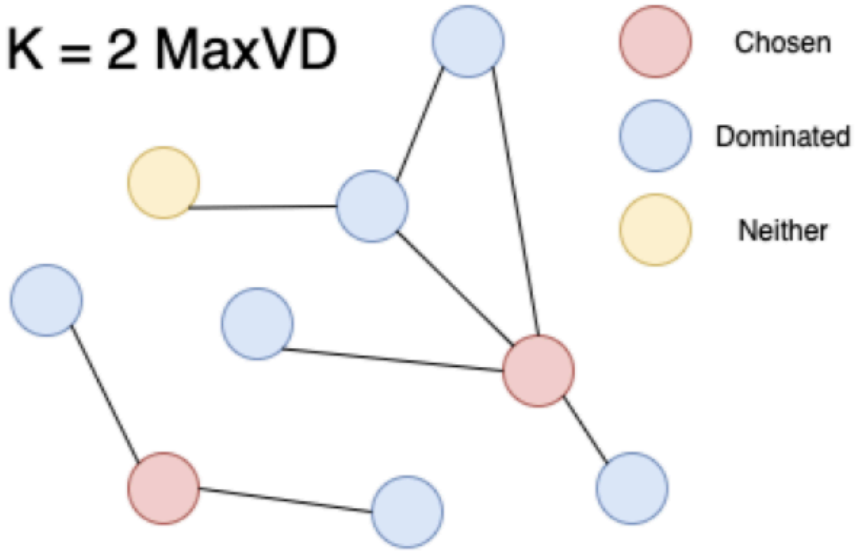
Task description (Domination and MDS)

- $G = (V, E)$
- A vertex v is said to dominate all its adjacent vertices and itself.
- A subset $D \subseteq V$ is a dominating set of G if every vertex in $V - D$ is adjacent to at least one vertex in D.
- a famous froblem minimum dominating set (MDS)
- have a lot of real-world applications



k-Maximum vertex domination(k-MaxVD)

- Now we consider a stronger variant of domination problem
- Given a graph $G = (V, E)$, an integer k, find the maximum number of vertices that can be dominated by a k-subset of V
- Note that one can solve the minimum dominating set problem by performing binary search on the number k. So the problem is NP-hard on general graphs.



RELATED WORKS

Related Works

- It is well-known that **finding a MDS** in general graphs is **NP-hard**.
- For one can solve MDS in polynomial time with a binary search on k with a polynomial time algorithm for k-MaxVD, we know **k-MaxVD is NP-hard**.
- Due to the submodularity property, A straight-forward greedy algorithm can achieve an approximation ratio of $1 - 1/e$ for k-MaxVD [2].
- A related research [1] shows some upper bounds of approximation ratio for the k-MaxVD problem.

Related Work(cont.)

- The problem in general graphs has shown its NP-hardness and a certain degree of inapproximability.
- However, there is a linear time algorithm for MDS on cactus graphs[3].
- So we believe the existence of a polynomial time algorithm for k-MaxVD problem on trees.

The main contribution

- In this paper, we devised a $O(k^2|V|)$ dynamic programming algorithm for solving k-MaxVD on trees
- We further improve the complexity to $O(k|V|)|)$ using SMAWK's algorithm [4].

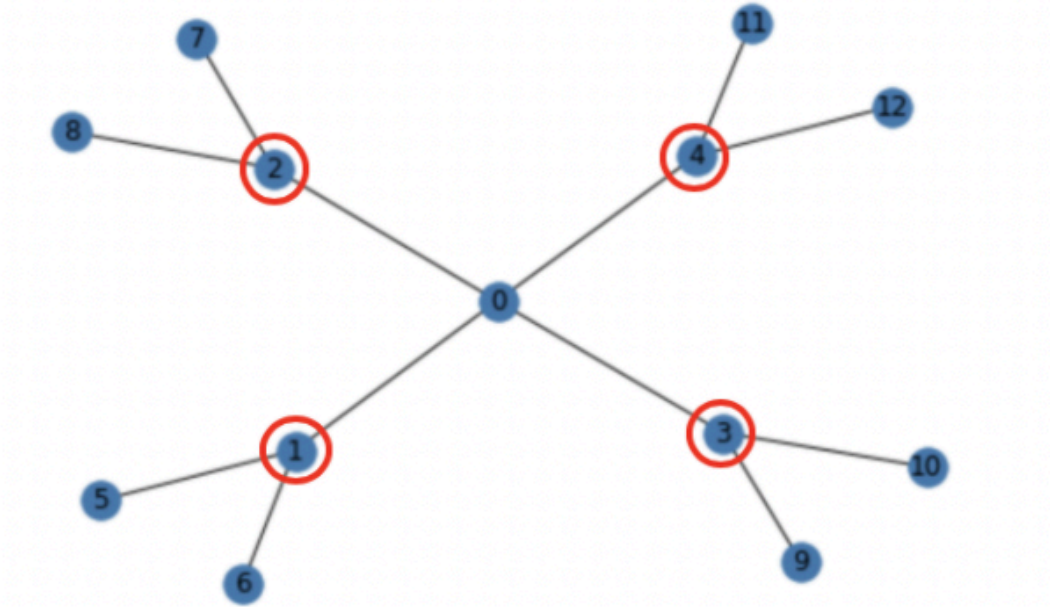
PROPOSED ALGORITHM

The Greedy Algorithm

- Iteratively chose vertex with max valid degree
- Maximum domination has **submodularity** property
- Greedy algorithm in trees can be done in $O(|V|)$
  - link vertex v with $degree(v)$, $N(v)$
    - can finding maximumam degree with amortized $O(|V|)$
    - $O(1) * O(|E|)$ modification
- $O(1 - 1/e)$-approximation

Greedy can be wrong

- Greedy will pick the **middle vertex at first step**
- While the unique solution does not contain it

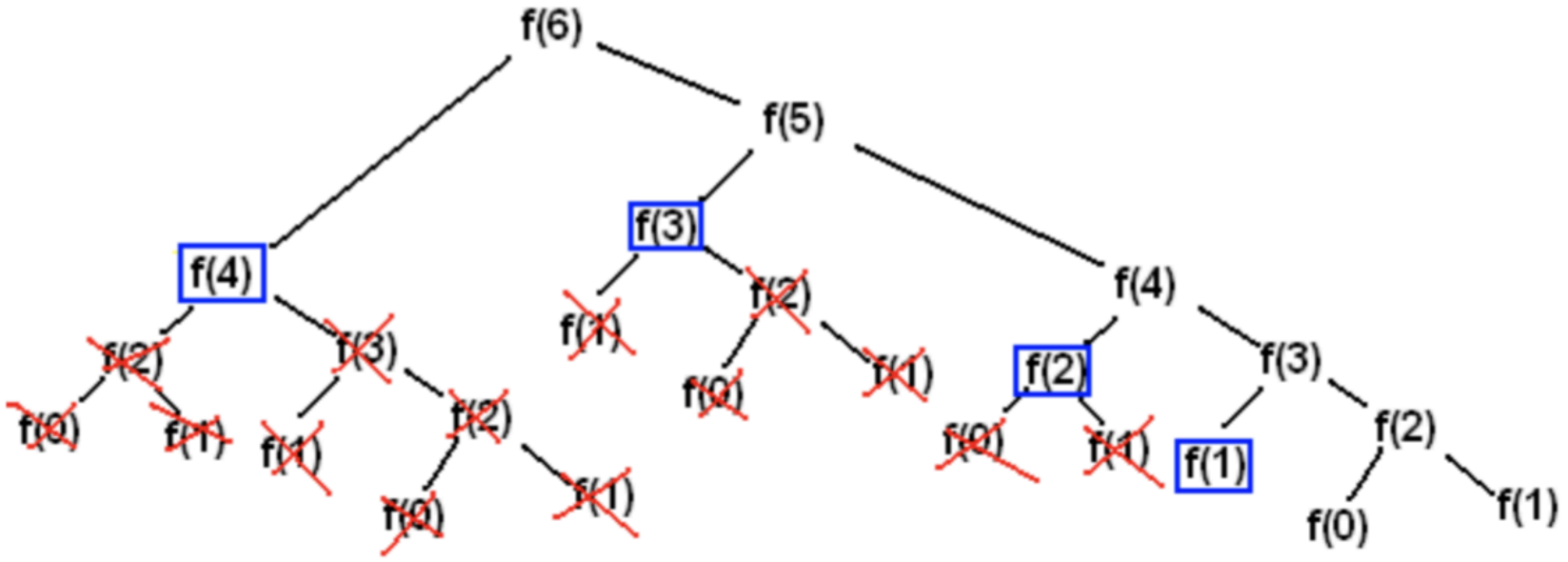


- An exhaustive search might be inevitible

THE VANILLA DP SOLUTION

Dynamic Programming 101

- We want to assume everyone have a basic knowledge about DP
- Fibonnaci and tree search



https://en.wikibooks.org/wiki/User:Gkhan/Fibonacci_using_memoization_and_DP

Motivation of using dynamic programming

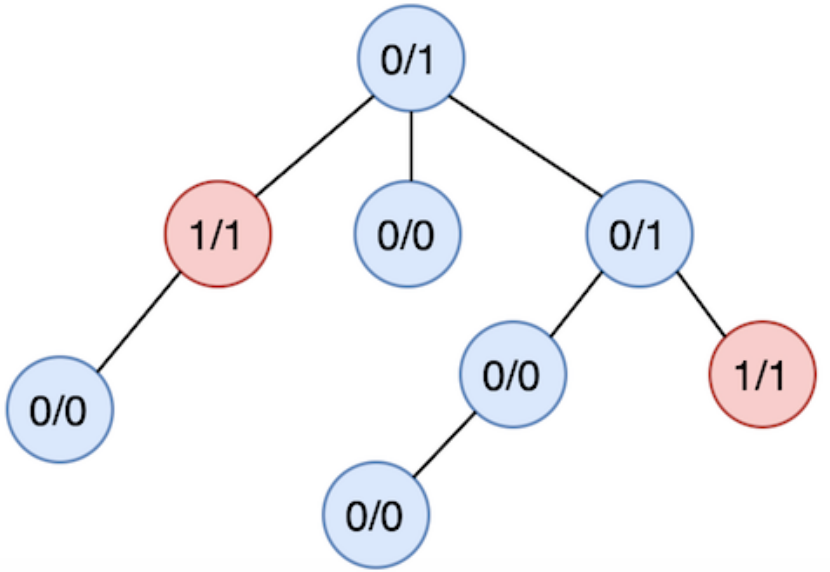
- As doing exhaustive search, one may noticed that the optimal solution of a subtree rooted at u (opt(u, k)) can be determined by those of u's children.
- This motivates us to think of the technique of memorization to avoid recalculations of children's result.

Transition (Part A)

First we do a depth first search (dfs) to make the tree rooted.

Denote $opt(u, k)(t, d)$ as the optiminal value that can be obtained when

- considering only the subtree rooted at u
- with k vertices to be allocated
- bit t means if the vertex u is selected
- bit d means if the vertex u is dominated.



Transition (part B)

For the base cases, if u is a leaf:

$$opt(u,k)(0,0) = 0$$
$$opt(u,k)(0,1) = -\infty$$
$$opt(u,k)(1,1) = \begin{cases} -\infty & \text{if } k \leq 0 \\ 1 & \text{otherwise} \end{cases}$$

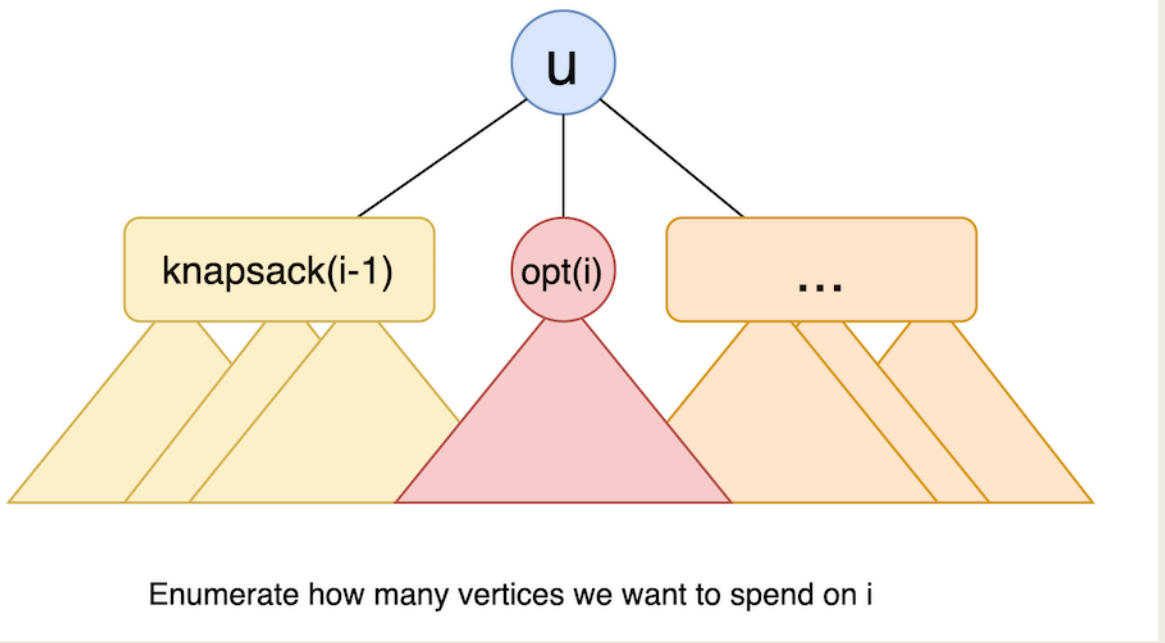
Transition (Part C)

For other cases:

$$opt(u,k)(0,0) = best(\{(0,0),(0,1)\})$$
$$opt(u,k)(0,1) = best(\{(0,0),(0,1),(1,1)\}) + 1 \ni \text{at least one } (1,1) \text{ is chosen}$$
$$opt(u,k)(1,1) = best(\{(0,0)+1,(0,1),(1,1)\}) + 1$$

Transition (Part D)

Suppose there are nc children of u, and the global parameter k is denoted as K.
Let $child(u, i)$ be the ith child of root u.
Let $knapsack(i, k)$ be the best value we can get after considering ith child of u and use k vertices
We calculate $knapsack(i, k)$ by trying all possible k_{cur} to use in the current subtree.



```
# for the case best({(0,0), (0,1))})
for i in 0..nc-1
    for k in 0..K
        for k_cur in 0..k
            option = knapsack(i-1, k-k_cur) +
                    max(opt(child(u,i),k_cur)(0,0),
                        opt(child(u,i),k_cur)(0,1))
            knapsack(i,k) = max(knapsack(i,k), option)
```

```
# for the case best({(0,0)+1, (0,1), (1,1)})+1
for i in 0..nc-1
    for k in 0..K
        for k_cur in 0..k
            option = knapsack(i-1, k-k_cur) +
            max(opt(child(u,i),k_cur)(0,0)+1,
                opt(child(u,i),k_cur)(0,1), opt(child(u,i),k_cur)(1,1))
            knapsack(i,k) = max(knapsack(i,k), option)
```

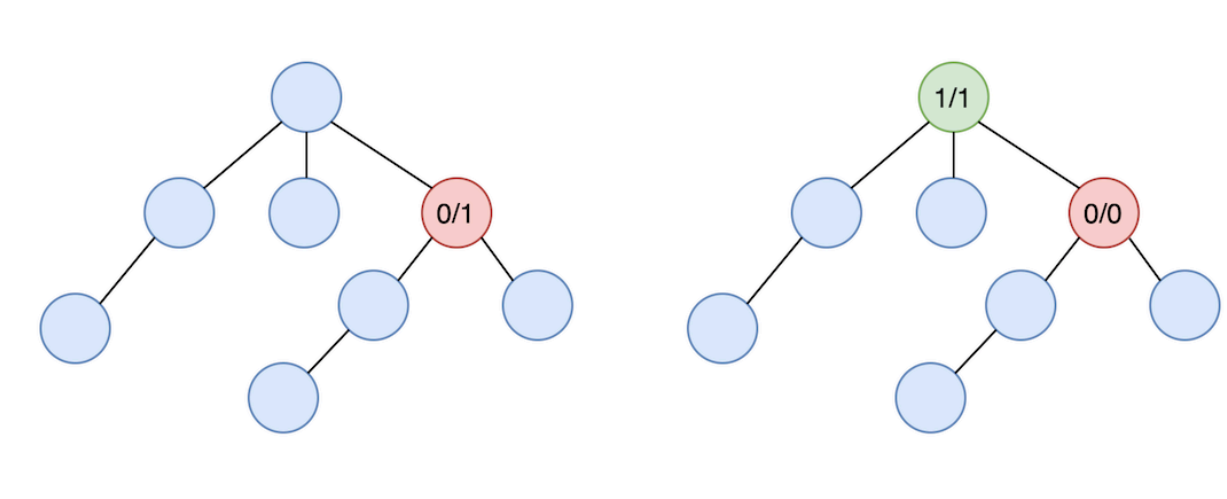
Transition (Part D) (cont.)

Notice that the case “at least one (1,1)” is more trickier, we have to maintain an extra bit in the knapsack to see whether the current configuration contrains an (1,1).

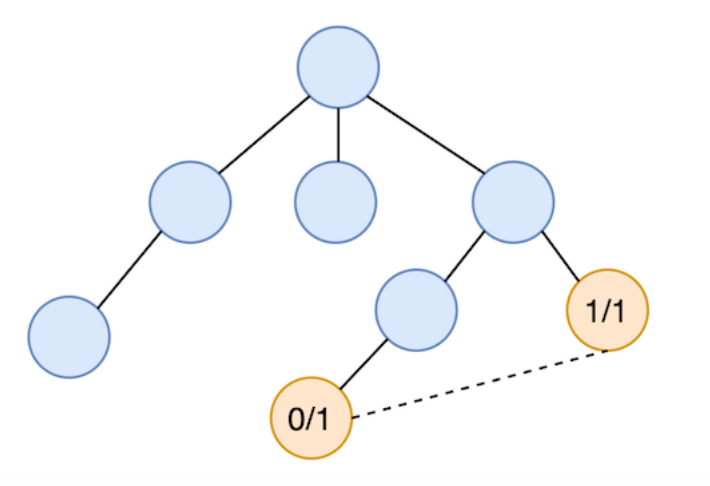
And we use the answer that considered all subtrees and has at least one (1,1) (i.e. $knapsack(nc-1,k)(1)$).

```
# for the case best({(0,0), (0,1), (1,1)})+1, at least one (1,1) is chosen
for i in 0..nc-1
    for k in 0..K
        for k_cur in 0..k
            val0 = max(opt(child(u,i),k_cur)(0,0), opt(child(u,i),k_cur)(0,1))
            val1 = opt(child(u,i),k_cur)(1,1)
            option0 = knapsack(i-1, k-k_cur)(0) + val0
            option1 = max(knapsack(i-1, k-k_cur)(0) + val1,
                          knapsack(i-1, k-k_cur)(1) + max(val0, val1))
            knapsack(i,k)(0) = max(knapsack(i,k)(0), option0)
            knapsack(i,k)(1) = max(knapsack(i,k)(1), option1)
```

Utility of the proposed algorithm by setting $-\infty$ to undesired states



To make the red dominated but not chosen
(This scenario is quite common in real world)
2 cases to try



To solve almost trees
About $2^{|cycle|}$ cases to try

Optimization with monge property

The transition treats the problem purely as allocating k resources to the subtrees, dropping the properties of our originally problem that are potentially useful.

By looking at the original problem more closely, we are actually finding row maximums in a matrix A s.t. $A[r][c] = knapsack(i-1)(r-c) + opt(i,c)(case)$ when transitioning from $i-1$ to i. SMAWK's algorithm[4] can speed the process from $O(k^2)$ to $O(k)$ if the A matrix is totally monotone, for more detailed description of SMAWK's algorithm, please refer to the original paper [4].

Optimization with monge property (cont.)

One may notice that the property of diminished return on allocating more resources to a subtree. More precisely,

$knapsack(i-1)(x+1) - knapsack(i-1)(x) <= knapsack(i-1)(y+1) - knapsack(i-1)(y)\ \forall y < x$

$option(x+1) - option(x) <= option(y+1) - option(y)\ \forall y < x.$

To prove for all pair (x, y) s.t. $y < x$, we only have to prove the pair $(x+1, x)$ and by induction the original satement is true.

If there is a case that $f(x+2) - f(x+1) > f(x+1) - f(x)$, suppose the choice from $f(x+1)$ to $f(x+2)$ is v. Than we can take v at the step from $f(x)$ to $f(x+1)$ and achieve a better $f(x+1)$, this contradicts the definition of $f(x+1)$.

Optimization with monge property (cont.)

We proceed to prove A is totally monotone.

Notice that monge property can implies totally monotone.

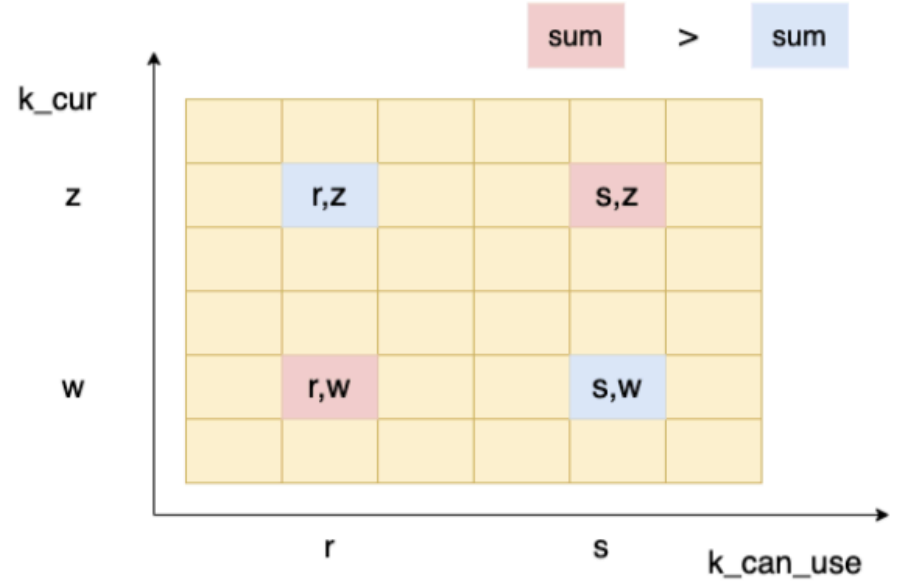
So it is suffice to show $A[r,w]+A[s,z] \geq A[s,w]+A[r,z]$ for all $w<z$ and $r<s$

$\forall w<z,\ r<s$

$$
\begin{aligned}
&A[r,w]+A[s,z] \\
&= knapsack(i-1)(r-w)+opt(i,w)(case)+knapsack(i-1)(s-z)+opt(i,z)(case) \\
&= knapsack(i-1)(r-w)+opt(i,w)(case)+knapsack(i-1)(r-z+(s-r))+opt(i,z)(case) \\
&\geq knapsack(i-1)(r-z)+opt(i,z)(case)+knapsack(i-1)(r-w+(s-r))+opt(i,w)(case) \\
&= A[s,w]+A[r,z]
\end{aligned}
$$

Now, with monge property and SMAWK's algorithm for finding row maximums, we can reduce the complexity of the transition from $O(k^2)$ to $O(k)$

Optimization with monge property (cont.)



$\forall w < z,\ r < s$

$$
\begin{aligned}
&A[r,w] + A[s,z] \\
&= knapsack(i-1)(r-w) + opt(i,w)(case) + knapsack(i-1)(s-z) + opt(i,z)(case) \\
&= knapsack(i-1)(r-w) + opt(i,w)(case) + knapsack(i-1)(r-z+(s-r)) + opt(i,z)(case) \\
&\geq knapsack(i-1)(r-z) + opt(i,z)(case) + knapsack(i-1)(r-w+(s-r)) + opt(i,w)(case) \\
&= A[s,w] + A[r,z]
\end{aligned}
$$

Pseudo Code

Algorithm 1 DFS

1: $p[u] \leftarrow f$;
2: $child[p] \leftarrow child[p] \cup \{u\}$;
3: push u to s;
4: **for** each $v \in N(u) \setminus \{p[u]\}$ **do**
5: $\quad DFS(v, u, p, s)$;
6: **end for**

Pseudo Code (cont.)

Algorithm 2 Maximum domination of k-vertex subset with tree DP

Input: the tree $T := (V, E)$, the number k;
Output: the maximum number of vertices s.t. a k-vertex subset can dominate;

$n \leftarrow |V|$;
$p \leftarrow$ an array of size n initialized with -1;
$s \leftarrow$ an empty stack;
$u \leftarrow$ the vertex with index 0 in T;
$child \leftarrow$ an array of set of size n initialized with $\emptyset$;

DFS$(u, -1, p, s)$;
// after DFS is performed, s contains the desired ordering for dynamic programming
$opt \leftarrow$ a 4-D integer array of size $n \times k \times 2 \times 2$ initialized with $-\infty$;
while $(s \neq \emptyset)$ **do**
  $u \leftarrow s.pop()$;
  if $(child(u) = \emptyset)$ **then**
    for $(i \leftarrow 0$ **to** $k)$ **do**
      $opt[u][i][0][0] \leftarrow 0$;
      $opt[u][i][0][1] \leftarrow -\infty$;
      $opt[u][i][1][1] \leftarrow (i >= 1)\ ?\ 1 : -\infty$;
    end for
    continue;
  end if
  $nc \leftarrow |child(u)|$
  $knapsack00 \leftarrow$ a 2-D integer array of size $|child(u)| \times k$
  $knapsack11 \leftarrow$ a 2-D integer array of size $|child(u)| \times k$
  $knapsack01 \leftarrow$ a 3-D integer array of size $|child(u)| \times k \times 2$
  initialize each item in $knapsack00, knapsack01, knapsack11$ with value $-\infty$
  // now we see set $child[u]$ as a 0-indexed array and do knapsack
  // for $opt[u][k][0][0]$
  for $(i \leftarrow 0$ **to** $nc - 1)$ **do**
    $v \leftarrow child[u][i]$
    for $(ki \leftarrow 0$ **to** $k)$ **do**
      for $(kj \leftarrow 0$ **to** $k)$ **do**
        $option \leftarrow (i > 0\ ?\ knapsack[i-1][ki-kj] : 0) + \max(opt[v][kj][0][0], opt[v][kj][0][1])$;
        $knapsack[i][ki] \leftarrow \max(knapsack[i][ki], option)$;
      end for
    end for
  end for
  for $(ki \leftarrow 0$ **to** $k)$ **do**
    $opt[u][ki][0][0] \leftarrow knapsack[nc-1][ki]$;
  end for
end while

Algorithm 3 Maximum domination of k-vertex subset with tree DP (cont.)

// for $opt[u][k][0][1]$
for $(i \leftarrow 0$ **to** $nc - 1)$ **do**
  $v \leftarrow child[u][i]$
  for $(ki \leftarrow 0$ **to** $k)$ **do**
    for $(kj \leftarrow 0$ **to** $k)$ **do**
      $val0 \leftarrow \max(opt[v][kj][0][0], opt[v][kj][0][1])$
      $val1 \leftarrow opt[v][kj][1][1]$
      $option0 \leftarrow (i == 0\ ?\ 0 : knapsack01[i-1][ki-kj][0]) + val0$;
      $option1 \leftarrow \max((i == 0\ ?\ 0 : knapsack01[i-1][ki-kj][0]) + val1, (i == 0\ ?\ -\infty : knapsack01[i-1][ki-kj][1] + \max(val0, val1)))$;
      $knapsack01[i][ki][0] \leftarrow \max(knapsack01[i][ki][0], option0)$;
      $knapsack01[i][ki][1] \leftarrow \max(knapsack01[i][ki][1], option1)$;
    end for
  end for
end for
$opt[u][0][0][1] \leftarrow -\infty$;
for $(ki \leftarrow 1$ **to** $k)$ **do**
  $opt[u][ki][0][1] \leftarrow knapsack01[nc-1][ki][1] + 1$;
end for
// for $opt[u][k][1][1]$
for $(i \leftarrow 0$ **to** $nc - 1)$ **do**
  $v \leftarrow child[u][i]$
  for $(ki \leftarrow 0$ **to** $k)$ **do**
    for $(kj \leftarrow 0$ **to** $k)$ **do**
      $option \leftarrow (i > 0\ ?\ knapsack[i-1][ki-kj] : 0) + \max(opt[v][kj][0][0] + 1, opt[v][kj][0][1], opt[v][kj][1][1])$
      $knapsack[i][ki] \leftarrow \max(knapsack[i][ki], option)$;
    end for
  end for
end for
$opt[u][0][1][1] \leftarrow -\infty$;
for $(ki \leftarrow 1$ **to** $k)$ **do**
  $opt[u][ki][1][1] \leftarrow knapsack[nc-1][ki-1] + 1$;
end for

Correctness of Algorithm

- The algotirhm, in essence, is doing enumaration on all possiblity of allocating resources on subtrees and all possible state for each subtree. While maintaining the information of the root of the subtree to complete the global problem.
- By using dynamic programming technique, we memorize the best results and avoid recalculation.
- It can be shown that all circumstances is considered at some stage of our dynamic programming algorithm.

Time Complexity Analysis

The Vanilla DP

- To discuss the conplexity of a dynamic programming algithm, we discuss the total transition time.
- We have $O(k^2 * deg(u))$ calculations in for each vertex u
- This sums up to $O(k^2 n)$ since $\sum_{u \in V} deg(u) = O(n)$
- So the total running time is $O(k^2 n)$.

SMAWK's algorithm

- Total run time is $O(kn)$.

EXPERIMENTAL RESULTS

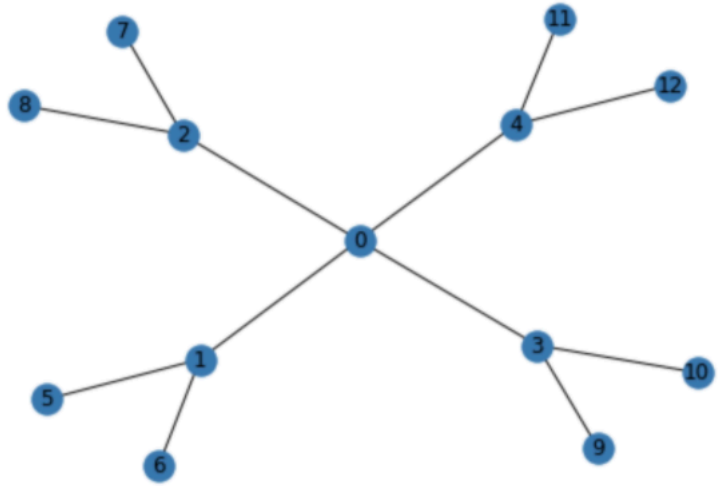
Code realization
Examples
Time Complexity Testing

Partial C++ Code

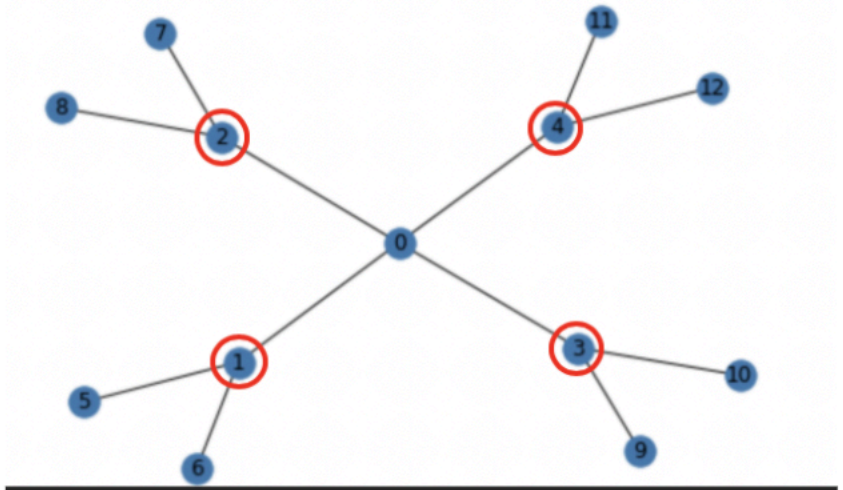
```
//dp[0][1]
vector<vector<int>> knapsack01[2];
knapsack01[0] = vector<vector<int>>(k+1, vector<int>(nc, -INF)); //-INF not 0
knapsack01[1] = vector<vector<int>>(k+1, vector<int>(nc, -INF)); //-INF not 0
rep(i, 0, nc)//order of visit
{
    int v = G[u][i];
    rep(ki, 0, k+1)//order of visit
    {
        rep(ki2, 0, ki+1) //now i use
        {
            int val0 = max(dp[v][ki2][0][0], dp[v][ki2][0][1]);
            int val1 = dp[v][ki2][1][1];
            int option0 = (i == 0 ? 0 : knapsack01[0][ki-ki2][i-1]) + val0;
            int option1 = max((i == 0 ? 0 :knapsack01[0][ki-ki2][i-1]) + val1,
                              (i == 0 ? -INF :knapsack01[1][ki-ki2][i-1] + max(val0,val1)));
            knapsack01[0][ki][i] = max(knapsack01[0][ki][i], option0);
            knapsack01[1][ki][i] = max(knapsack01[1][ki][i], option1);
        }
    }
}
```

Example 1 – Handcraft testcase

$n = 13, k = 4$



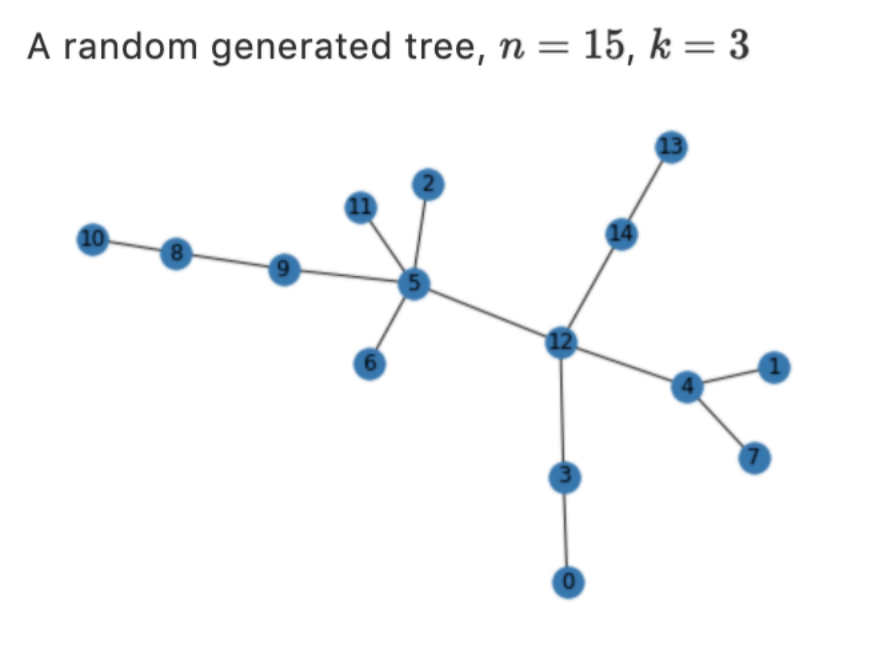
Lin's: 13
greedy : 12

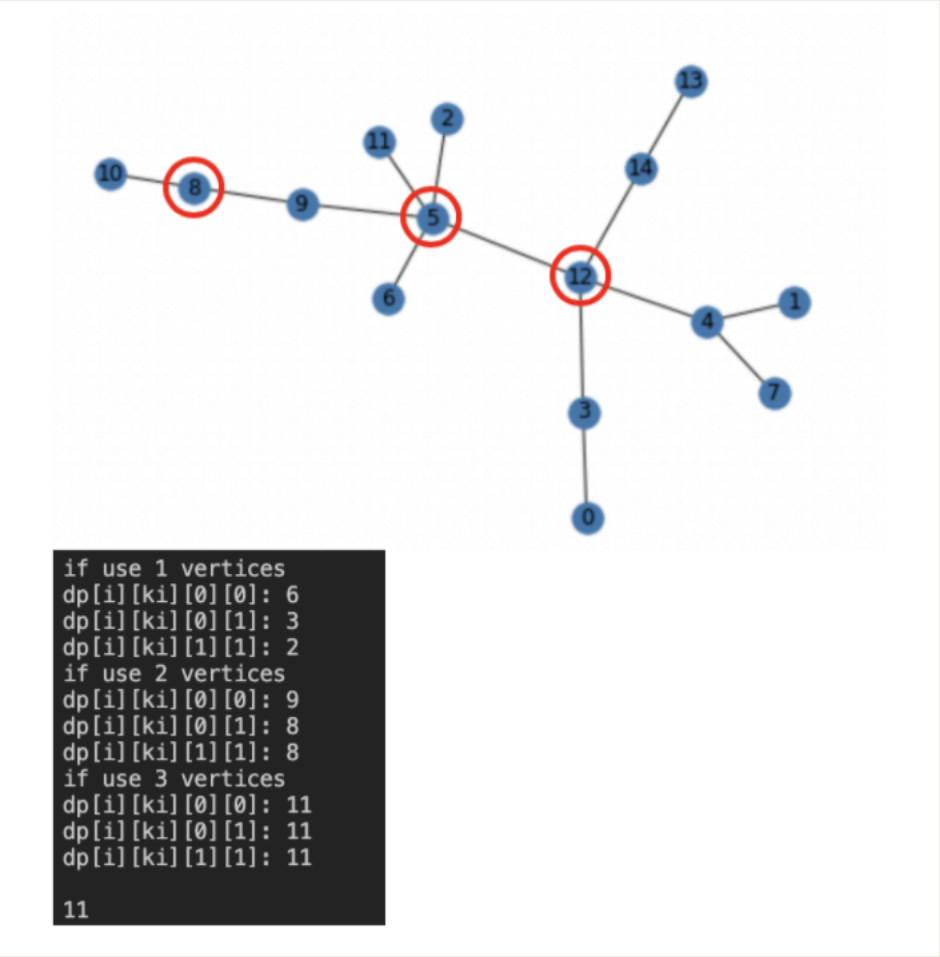



```
if use 2 vertices
dp[i][ki][0][0]: 4
dp[i][ki][0][1]: 7
dp[i][ki][1][1]: 7
if use 3 vertices
dp[i][ki][0][0]: 6
dp[i][ki][0][1]: 10
dp[i][ki][1][1]: 9
if use 4 vertices
dp[i][ki][0][0]: 8
dp[i][ki][0][1]: 13
dp[i][ki][1][1]: 11

13
linyantongs-MacBook-Pro:Individual-Study-AM-2020-spring maxwill$
```

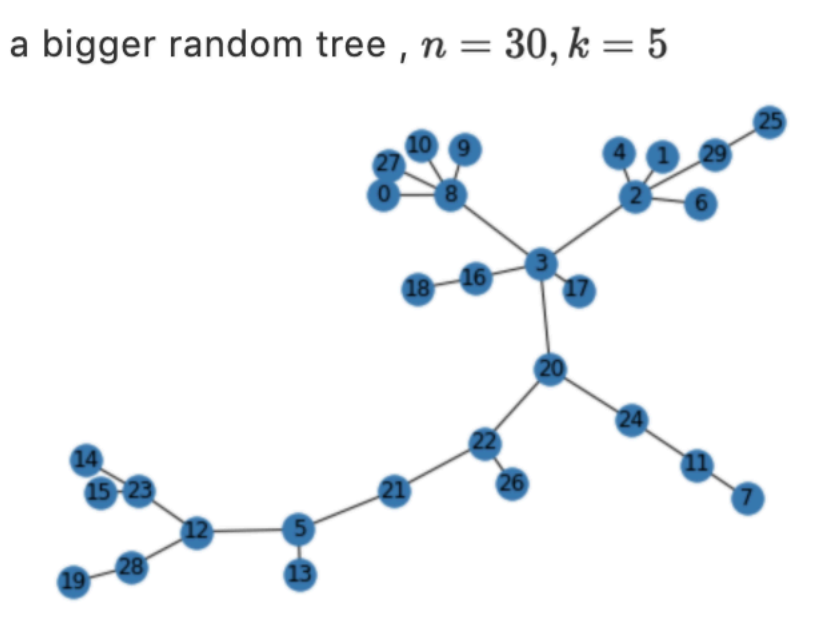
Example 2 – Random Small

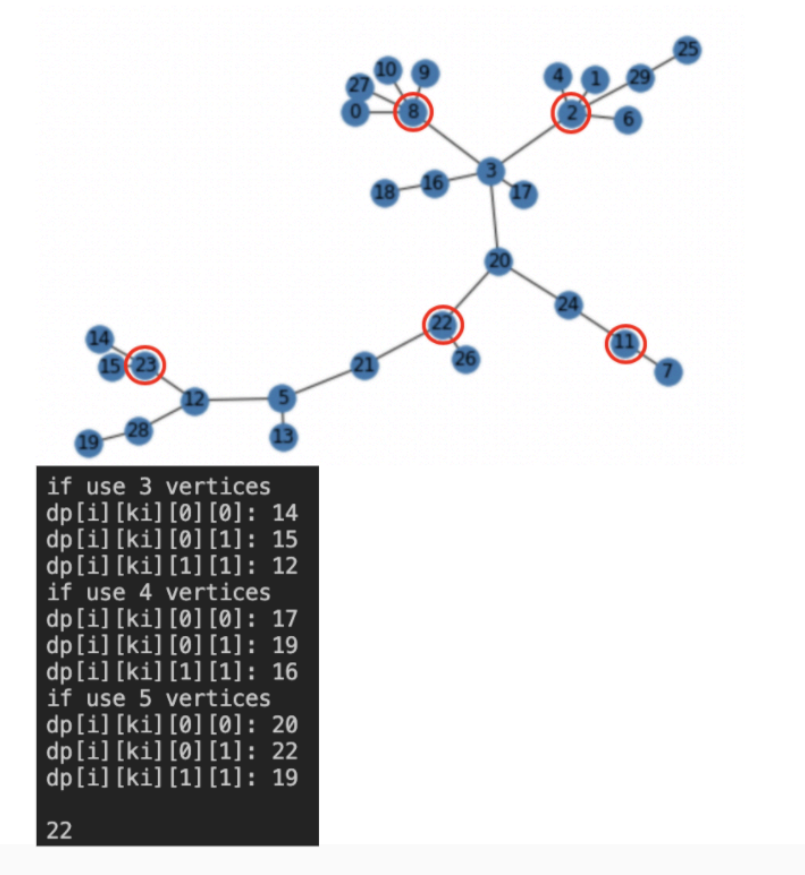




Example 3 – Random Large

- A truly large (ex: 10000+ nodes) test case is not easy to visualize, so we use a smaller one to demo
- But we have verified that our solution is correct on a larger datum

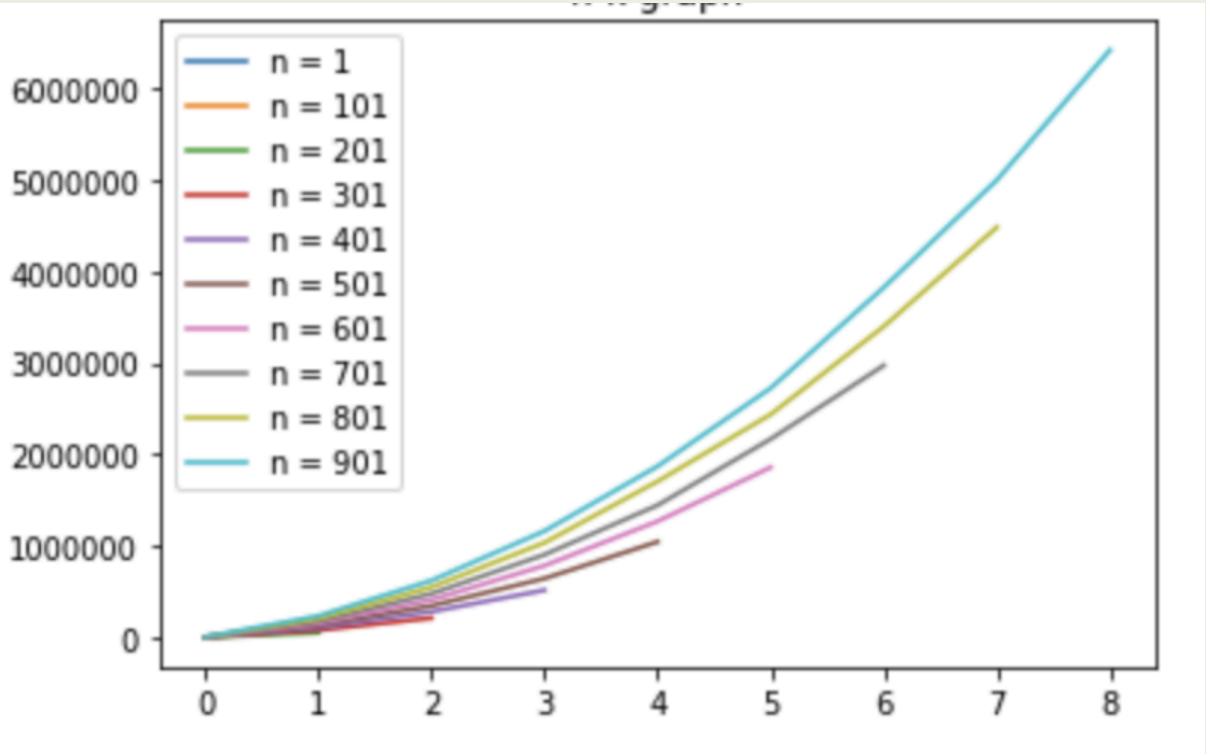




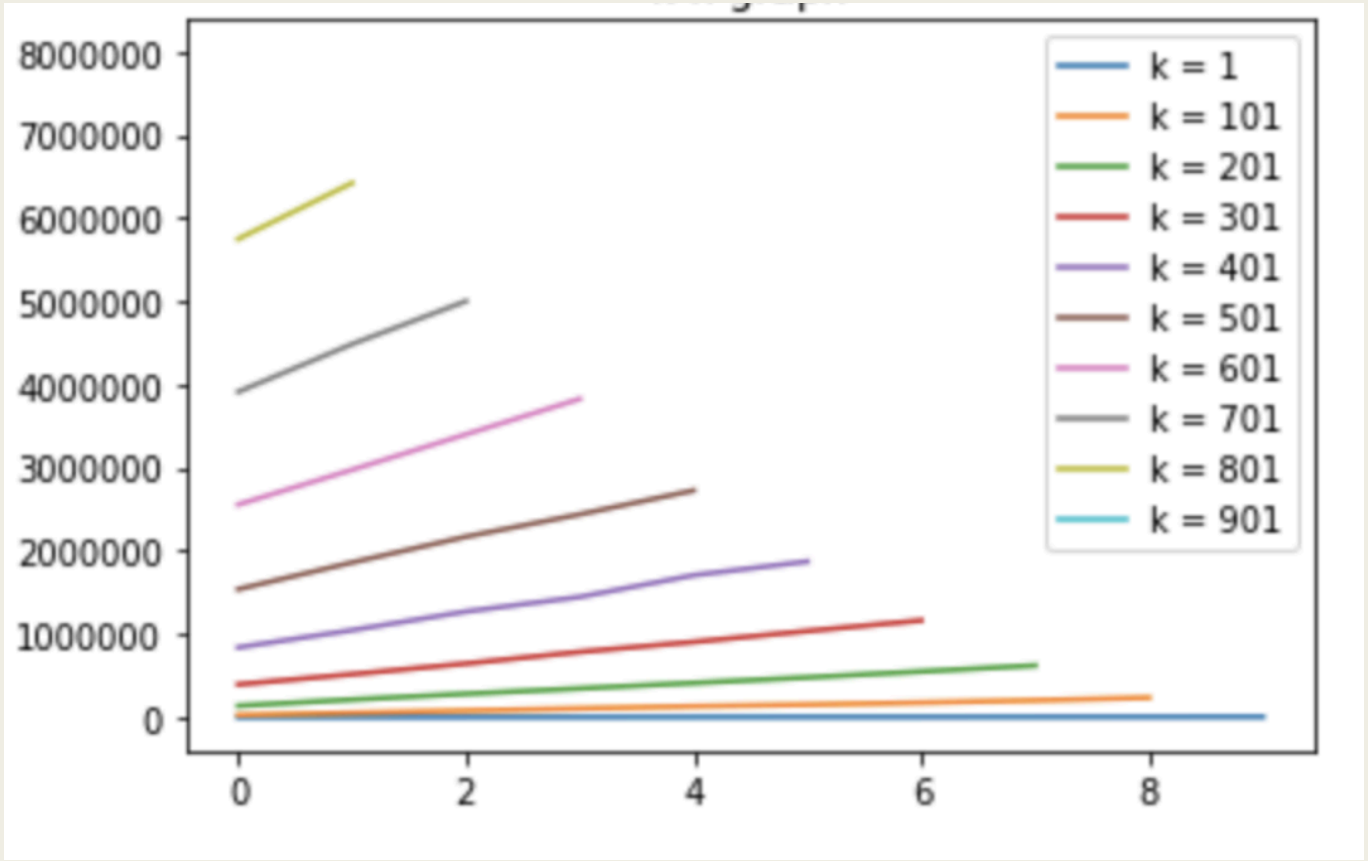
Time Complexity Testing

- How is experiment done
  - g++ compiles c++ program
  - python program generates testcase with networkx
  - python program uses subroutine to run c++ execution file
  - python program plots the results
  - done on macbook pro 19
    - processor: 1.4 GHz Intel Core i5
    - operating system: macOS Mojave Version 10.14.6
- Experinent setting
  - n range from 1 to 901
  - k range from 0 to n
- Measurement
  - **sum of clocks** for each testcase's running time(exluding IO)

K-T graph


n = 1
n = 101
n = 201
n = 301
n = 401
n = 501
n = 601
n = 701
n = 801
n = 901
6000000
5000000
4000000
3000000
2000000
1000000
0
0
1
2
3
4
5
6
7
8

N-T graph


8000000
7000000
6000000
5000000
4000000
3000000
2000000
1000000
0
0
2
4
6
8
k = 1
k = 101
k = 201
k = 301
k = 401
k = 501
k = 601
k = 701
k = 801
k = 901

Concluding Remark

In this paper, we proposed a vanilla dynamic programming algorithm that solves k-MaxVD in trees with time complexity $O(k^2|V|)$ and further reduce the complexity to $O(k|V|)$ with SMAWK's algorithm.

The Vanilla DP can be apllied in a wider range if use the set-INF technique properly.

One remainning work is to prove that any algorithm that solves k-MaxVD in trees runs in $\Omega(k^2|V|)$ time or has a lower lower bound.

Reference

[1] Miyano, E., Ono, H.: Maximum domination problem. In: Proceedings of the Seventeenth Computing: The Australasian Theory Symposium, vol. 119, pp. 55–62. Australian Computer Society Inc (2011)
[2] S. Fujishige, Submodular Functions and Optimization, Annals of Discrete Math., vol. 47, 1990.
[3] Hedetniemi S.T., Laskar R., Pfaff J.: A linear algorithm for finding a minimum dominating set in a cactus, Discrete Appl. Math., 13 (2–3) (1986), pp. 287-292
[4] Peter Shor, Shlomo Moran, Alok Aggarwal, Robert Wilber, and Maria Klawe: Geometric applications of a matrix-searching algorithm. Algorithmica, 2:195–208,
1987.

Acknowledgement

- I would like to thank Dr. Chiuyuan Chen, my instructor of individual study, for giving me a lot of advices and help me come up with interesting ideas.

- I would also like to thank Dr. Meng-Tsung Tsai, my teacher of Advanced Algorithms, for encouraging me to utilize the knowledge in the class.

- And thank you for listening!