Maximum domination of k-vertex subset in trees

Yan-Tong Lin(林彥彤) (0312fs3@gmail.com)

Department of Applied Mathematics, National Chiao Tung University, Taiwan

Advisor: Prof. Chiuyuan Chen (陳秋媛)

Abstract

The minimum dominating set (MDS) problem has been widely studied and has numerous applications in computer networks. Here we consider a variation of MDS, the maximum domination problem (k-MaxVD). Given a positive integer k, find the maximum number of vertices that can be dominated by a k-vertex subset. The problem is NP-hard since one can solve MDS, an NP-hard problem, with a binary search on the parameter k. And the best approximation ratio of k-MaxVD problem is shown to be (1−1/e) unless P=NP. In this paper, we provide a polynomial dynamic programming solution for k-MaxVD in trees in $O(k^2|V|)$ time. And optimize the time complexity to O(k|V|) by the monge property of the dynamic programming table and SMAWK's algorithm [4].

Introduction

Terminologies regarding to the domination problem:
Let G = (V,E) be a graph.
N(v) of a vertex v is defined to be the set of all vertices adjacent to v.
And we define N[v], the closed neighborhood of v, to be N(v)∪{v}.
For any subset of V, named S, $N(S) := \cup_{v \in S} N(v)$ and $N[S] := \cup v \in S\ N[v]$
A vertex v is said to dominate all vertices in N[v].
A subset D⊆V is a dominating set of G if every vertex in V−D is adjacent to at least one vertex in D.

Definition of the k-MaxVD Problem:
Given a graph G = (V,E) and a positive integer k, find the maximum number of vertices in G that can be dominated by a k-vertex subset of V.

Previous works:
It is well-known that finding an 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 straightforward greedy algorithm can achieve an approximation of 1-1/e for k-MaxVD [2]. A related research [1] shows some upper bounds of the approximation ratio for the k-MaxVD problem.

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

Main contributions:
In this study, we develop a $O(k^2|V|)$-time dynamic programming algorithm for solving k-MaxVD on trees, and we further improve the complexity to O(k|V|) by using SMAWK's algorithm [4].

Purposed Algorithm

Here we give the transitions of our dynamic programming solution.
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, the bit t means if the vertex u is selected, and the bit d means if the vertex u is dominated.

For the base cases, if u is a leaf:

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

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

The $best(S)$ is the optimal value that we can get by distributing the k (or k-1, if the root u uses one quota) available vertices to subtrees of u, with the cases of the subtrees' roots are in S. This can be done by an analog to the well-known knapsack algorithm.

Algorithm 1 DFS

```
p[u] ← f;
child[p] ← child[p] ∪ {u};
push u to s;
for each v ∈ N(u) \ {p[u]} do
    DFS(v, u, p, s);
end for
```

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

```
// for opt[u][k][0][1]
for (i ← 0 to nc − 1) do
    v ← child[u][i]
    for (ki ← 0 to k) do
        for (kj ← 0 to ki) do
            val0 ← max(opt[v][kj][0][0], opt[v][kj][0][1])
            val1 ← opt[v][kj][1][1]
            option0 ← (i == 0 ? 0 : knapsack01[i − 1][ki − kj][0]) + val0;
            option1 ← max((i == 0 ? 0 : knapsack01[i − 1][ki − kj][0]) +
val1, (i == 0 ? −∞ : knapsack01[i − 1][ki − kj][1] + max(val0, val1)));
            knapsack01[i][ki][0] ← max(knapsack01[i][ki][0], option0);
            knapsack01[i][ki][1] ← max(knapsack01[i][ki][1], option1);
        end for
    end for
end for
opt[u][0][0][1] ← −∞;
for (ki ← 1 to k) do
    opt[u][ki][0][1] ← knapsack01[nc − 1][ki][1] + 1;
end for
// for opt[u][k][1][1]
for (i ← 0 to nc − 1) do
    v ← child[u][i]
    for (ki ← 0 to k) do
        for (kj ← 0 to k) do
            option ← (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] ← max(knapsack[i][ki], option);
        end for
    end for
end for
opt[u][0][1][1] ← −∞;
for (ki ← 1 to k) do
    opt[u][ki][1][1] ← knapsack[nc − 1][ki − 1] + 1;
end for
```

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 ← |V|;
p ← an array of size n initialized with −1;
s ← an empty stack;
u ← the vertex with index 0 in T;
child ← an array of set of size n initialized with ∅;

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

Optimization with monge property

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

By looking at the 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].

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 $f(x+1)$.

We proceed to prove A is totally monotone.
Notice that monge property can implies totally monotone.
So it is suffice to show $A[r,w] + A[s,z] \ge A[s,w] + A[r,z]$ for all $w < z$ and $r < s$

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

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

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 $O(k)$

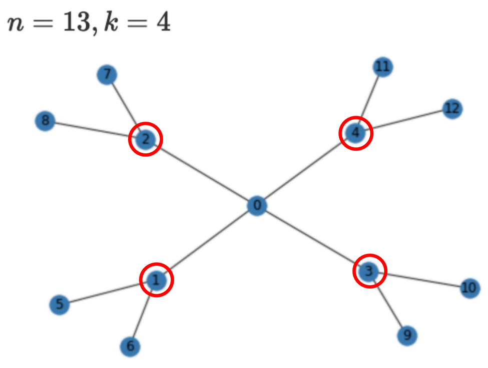
Experimental results

♦ Demo of the proposed algorithm running on a handcraft testcases and some bigger random testcases

Example 1

A hand craft testcase that the greedy algorithm can fail by choosing the vertex with maximum degree at the first step

$n = 13, k = 4$

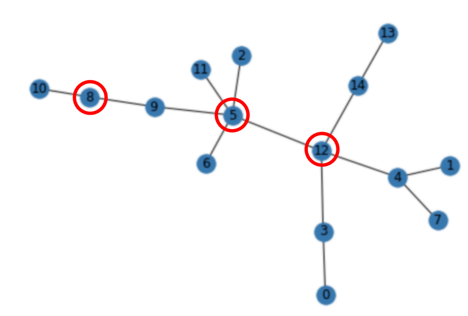


```
if use 2 vertices
dp[i][ki][0][0]: 4
dp[i][ki][0][1]: 7
dp[i][ki][1][1]: 7
if use 3 vertices
dp[i][ki][0][0]: 6
dp[i][ki][0][1]: 10
dp[i][ki][1][1]: 9
if use 4 vertices
dp[i][ki][0][0]: 8
dp[i][ki][0][1]: 13
dp[i][ki][1][1]: 11

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

Example 2

A random generated tree, $n = 15$, $k = 3$

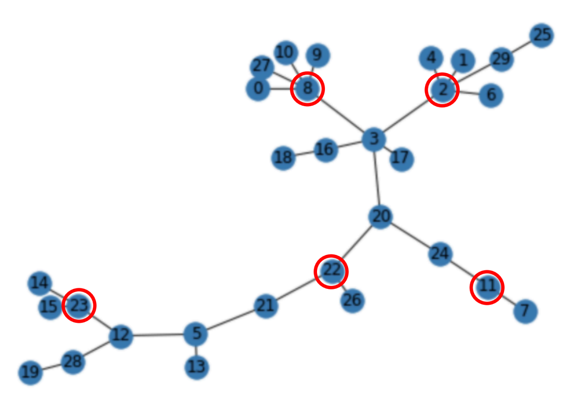


```
if use 1 vertices
dp[i][ki][0][0]: 6
dp[i][ki][0][1]: 3
dp[i][ki][1][1]: 2
if use 2 vertices
dp[i][ki][0][0]: 9
dp[i][ki][0][1]: 8
dp[i][ki][1][1]: 8
if use 3 vertices
dp[i][ki][0][0]: 11
dp[i][ki][0][1]: 11
dp[i][ki][1][1]: 11

11
```

Example 3

a bigger random tree , $n = 30, k = 5$



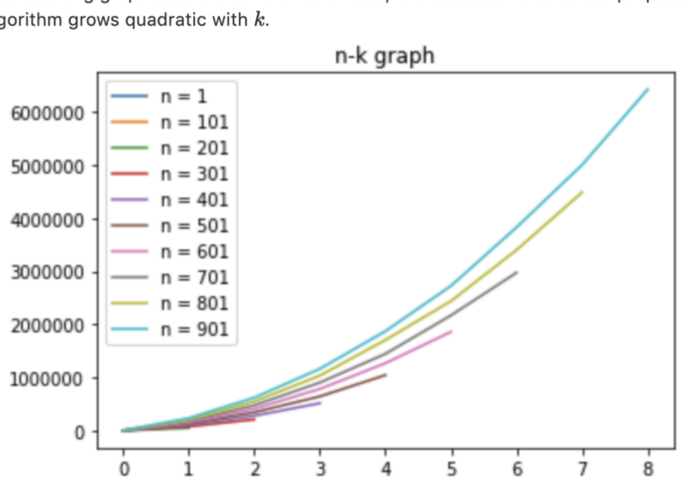
```
if use 3 vertices
dp[i][ki][0][0]: 14
dp[i][ki][0][1]: 15
dp[i][ki][1][1]: 12
if use 4 vertices
dp[i][ki][0][0]: 17
dp[i][ki][0][1]: 19
dp[i][ki][1][1]: 16
if use 5 vertices
dp[i][ki][0][0]: 20
dp[i][ki][0][1]: 22
dp[i][ki][1][1]: 19

22
```

♦ Experiments on the time complexity of the Vanilla DP algorithm

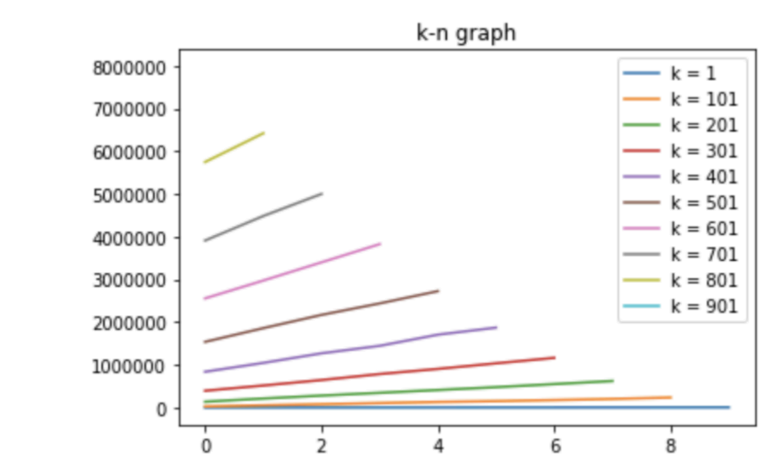
n-k graph

- The resulting graph shows that when n is fixed, the execution time of our proposed algorithm grows quadratic with k.



k-n graph

- the resulting graph shows that when k is fixed, the execution time of our proposed algorithm is linear to n
- and the slope is related with k



Concluding remarks

In this study, we propose 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|) by SMAWK's algorithm.

One future work is to prove that any algorithm that solves k-MaxVD in trees requires Ω(k|V|) time or has a lower lower bound.

References

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

I would like to thank Prof. Chiuyuan Chen, my instructor of this individual study, for giving me a lot of advices and helping me come up with interesting ideas.
I would also like to thank Prof. Meng-Tsung Tsai, my teacher of the course Advanced Algorithms, for encouraging me to utilize the knowledge in the course.