CS787: Advanced Algorithms

Topic: Fixed parameter algorithms **Presenter(s):** Noah Larsen, Hesam T. Dashti, David Guild

17.7.1 Introduction

Understanding parameterization begins with the idea of decomposing a computational problem into additional parameters, and viewing the amounts of resources used by algorithms for such problems as a function of not just the length of the input, but of other the other parameters as well. This is done, for the purposes of complexity theory, to try to understand in terms of explicit parameters just what makes a hard problem hard, and for the purposes of algorithm design, to try to develop algorithms for hard problems that can be considered tractable for large inputs given certain parameters. This is not merely algorithm design for certain subclasses of inputs, as parameters form a cartesian product with the input space, instead of forming a strict subset that can be shown to be solvable in polynomial time. The desired algorithms here are ones in which the exponential hardness of the problem can be relegated in totality to a particular parameter as opposed to the length of the input. The potential usefulness of these algorithms are that, in the same way that traditional exponential running time algorithms are still tractable for all inputs of small length, for small values of the parameters, the algorithms here will be tractable for inputs of the same ranges of lengths as are tractable for traditional polynomial time algorithms - virtually all lengths that occur in practice. This phenoma can be further contrasted with algorithms for certain subclasses of inputs in that there the difference between tractability and intractability is sharp, as for certain inputs the problem is solvable in polynomial time and in general it is not known to be, whereas here the difference between tractability and intractability is a matter of degree.

Parameterized Problem
A paramaterized problem is a language L $\subseteq\{\sum^* \times \sum^*\}$
The parameter is usually a nonnegative integer.

Fixed-Parameter Tractable
A problem is fixed-parameter tractable if there exists an algorithm that decides it that runs in f(k)p(|x|) time, where k is the problem's parameter, f is some function depending only on k, and p is some polynomial function. Notice that while the running time must be polynomial in the size of the input, it does not have to polynomial in the parameter. FPT is the complexity class of such problems.

Kernelization

One of the primary fixed-parameter algorithm design techniques is known as kernelization, and can be informally thought of as data reduction/preprocessing rules that have provable performance

guarantees. It is defined as follows.

Let L be a parameterized problem, that is, L consists of input pairs (I, k), where I is the problem instance and k is the parameter. A reduction to a problem kernel (or kernelization) means to replace an instance (I, k) by a reduced instance (I',k') called problem kernel such that
(1) k' = k,
(2) the size of I' is smaller than g(k) for some function g only depending on k and
(3) (I, k) has a solution if and only if (I',k') has one. The reduction from (I, k) to (I',k') must be computable in polynomial time.

Theorem. A problem is fixed-parameter tractable if and only if it has a kernelization.

The most important aspect of this theorem is why if a problem has a kernelization it is fixed-parameter tractable, and is precisely the reason why kernelizations can be used in the design of fixed-parameter tractable algorithms. Such an algorithm can be created by reducing an instance using the reduction provided by the kernelization, and then solving the reduced parameterized input with some sort of brute force algorithm. As the size of the reduced instance is bounded by the original parameter, the brute force algorithm will run in time exponential in that parameter. The smaller the size of the reduced instance is with respect to k, the more efficient the algorithm.

Example of a Kernelization for Vertex Cover
Consider the following parameterized version of Vertex Cover: given a graph, does there exist a vertex cover consisting of k or fewer vertices? Below is a simple kernelization for vertex cover.
Reduction Rule 1: Remove all isolated vertices.
Reduction Rule 2: For vertices with a degree of one, put the neighboring vertex into the cover.
Reduction Rule 3: Put any vertex that has a degree of at least k+1 into the cover.

Because any vertice in the reduced graph has degree of at most k, and the solution has to have at most k vertices, the reduced graph can have a solution only if it has at most k^2 edges. Because of this and the fact that every vertice in the reduced graph will have degree at least two, the reduced graph can have a solution only if it has at most k^2 vertices. The size of the reduced graph is thus k^2.

17.7.2 A Parameterized Algorithm for the scheduling problem with precedence constraints on a single machine

The scheduling problem for a collection of weighted tasks on a single machine with respect to a set of given precedence constraints is an NP-Hard problem [2]. The problem prototype is as follows:
A set of tasks: $T=\{t_1, t_2, \ldots, t_n\}$
A set of weights $W=\{w_1, w_2, \ldots, w_n\}$
A set of processing times $P=\{p_1, p_2, \ldots, p_n\}$

A set of ending times C=$\{c_1, c_2, \ldots, c_n\}$
The aim: minimize $\sum_{i=1}^{n} c_i w_i$
Lets consider the given constraints as a directed acyclic graph (G(V, E)) which is represented by an adjacency matrix M.
In order to propose a fixed parameter algorithm, we must extract some preliminary information about the tasks.

17.7.2.1 Preliminaries:

Based on the precedence matrix M, we can construct a depth(layer) tree which represents the priority in considering the nodes; e.g. nodes with no-input are in depth 0, and nodes which just depend on the nodes in layer 0, are located in depth 1. in layer 2 we have nodes that depend on the nodes in layer 1 and/or layer 0. Generally, the nodes in layer i depend on the nodes in layer $i-1$ and/or $i-2 \ldots 0$).
Let L be the maximum number of nodes in the layers. The value of L can be calculated by a polynomial function via tracing entries of the M matrix where entries are 1 for in-edges and -1 for the out-edges.
Let K denote the maximum number of out-degree in the matrix M. This value is easily computable since maximum absolute value of sums of '-1' in each row equals to K. An example of precedence graph with its corresponding layers is indicated in Figure 17.7.2.1. In the next section we focus on parameterizing the scheduling problem.

Figure 17.7.2.1: This figure represents different layers of the precedence graph of a sample scheduling problem.

17.7.2.2 Parameterizing the scheduling problem

As described in the introduction, in order to parameterize a problem we need a "Kernel" function which redefines the problem into a problem with polynomial time complexity on magnitude of a given input.

Kernelization

Rule 1. If we have 1 task with no-constraint (t_i), Problem(T) = Problem(T/t_i). Subsequently all out-edges of t_i should be removed.
Rule 2. Let S denotes the tasks without any precedent. If the cardinality of S is more than 1, a backtrack function will provide the decision of choosing first priority for a member of S. In order to calculate complexity order, let all the layers includes (EL) nodes, which is the worst case (provided that each layer has the maximum probable of nodes). In order to calculate the maximum probable nodes in each layer, assume a layer which includes L nodes, one of them (that has high priority) has K branches and each of them K branch and so on. In the worst case we will have E layers, so at the last layer we will have EL nodes. To find first node in layer i, we have to call following backtrack function for all the nodes in this layer. The backtrack function continues along the graph until it reaches the last layer where there is not any precedence node. All the nodes in this layer are ready to be considered. In this case, we sort the nodes based on $\frac{p_i}{w_i}$ and consider them in descending order. The number of layers that must be considered for each node in layer i is the depth of the tree subtracted by i. For simplicity in calculating the complexity order we could use the depth of precedence three for all layers (without subtracting i). The maximum depth of a precedence graph is equal to number of out-edges (the E set) (e.g. in a linear graph). We use this value as worst case of the depth for all layers.
We conclude that the order of choosing a node from a layer of precedence is equal to $(EL)^E$. We have to repeat this process for all nodes; therefore, the complexity order will be
O($(EL)^E$).O(n) = O($(EL)^E * n$).

```
Function Sln(T, E)   % T: nodes in current layer, E: the set of edges of the precedence matrix
begin
  For all v_i ∈ T
    Temp = T
    weight_i[w] = w_i    % weight_i has two entries for storing w and c
    weight_i[c] = c_i    % weight_i has two entries for storing w and c
    weight_i += Sln(Temp/v_i ∪ {S ∈ T} , E/uv_i{uv_i ∈ E})
    Temp = S
  choose v_i with minimum w.c
  return weight_i
end.
```

Hence, the complexity order of the problem is changed into 'n' times of $(EL)^E$. The exponential part of the complexity is not directly related to 'n'. For any given sample of the problem, K, L, and E can be calculated in polynomial time complexity.

17.7.3 Clique Partition

For a given graph G, a *clique cover* is defined to be a set of cliques that cover all the edges of G. The Clique Cover problem simply asks for the smallest such set. We are going to look at a slightly different problem, Clique Partition, that requires all the chosen cliques to be edge disjoint.

The parameterized version of Clique Partition is as follows.

Given: A graph $G = (V, E)$ and an integer k

Return: *Yes* if there is a set of at most k cliques in G such that every edge in E belongs to exactly one such clique, and *no* otherwise.

We will show a reduction that uses four rules to find a graph of at most k^2 vertices.

- **Rule 1:** For any vertex v of degree zero, remove it from the graph.
- **Rule 2:** For any vertex v of degree one, add the clique of $\{v, N(v)\}$ to the solution and remove v from the graph.
- **Rule 3:** For any edge $e = (u, v)$ such that $N(u)$ and $N(v)$ are disjoint, add the clique of $\{u, v\}$ to the solution and remove e from the graph.

This next theorem is by de Bruin and Erdos. We are most interested in the following lemma.

Theorem 17.7.3.1 *Suppose $A_1, \ldots, A_m$ are subsets of the set $A = \{_a1, \ldots, a_n\}$, and that $A_i \neq A$, $1 \leq i \leq m$. If each pair $\{a_r, a_s\}$ occurs in one and only one A_i, then $m \leq n$, and equality holds if and only if either*

1. *$A_1 = \{a_1, \ldots, a_{n-1}\}$, $A_2 = \{a_1, a_n\}$, $\ldots$, $A_n = \{a_{n-1}, a_n\}$, or*
2. *n is of the form $n = k(k+1)+1$ and all the A_i s have precisely $k+1$ elements, and each a_j occurs in exactly $k+1$ of the A_is, $1 \leq i \leq m, 1 \leq j \leq n$.*

Lemma 17.7.3.2 *Let G be a graph of size k. Any nontrivial clique partition of G must have at least k cliques.*

Lemma 17.7.3.3 *Let S be a clique partition of a graph G of size k. If $G' \subseteq G$ is a complete subgraph of G on more than k vertices, then there is an element $C \subseteq S$ such that $G' \subseteq C$.*

This leads us to the fourth reduction rule.

- **Rule 4:** Suppose v is a vertex of degree at least k, and that the neighbors of v form a clique. Then the subgraph G^* induced by v and its neighbors is a clique of size greater than k. Add this clique to the solution and remove its edges and the vertex v from the graph.

If the original problem is a yes-instance, then by the lemma above there must be some clique in the solution that contains the clique we removed with rule 4. However, the construction of that clique ensures that it is maximal, so for any solution to the unreduced problem, any clique found with rule 4 must be an element of the solution. Then the reduced problem is a yes-instance if the original problem was.

The converse is even simpler. If the reduced problem is a yes-instance (with a suitably modified k value), then clearly adding the clique found in rule 4 will create a yes-instance of the unreduced problem.

An instance of Clique Partition is fully reduced if none of the four reduction rules can be applied.

Theorem 17.7.3.4 *If a fully reduced instance G of Clique Partition has a solution of size at most k, then it has at most k^2 vertices.*

Let S be a set of cliques that partitions G. Consider a clique $C \in S$ that contains more than k vertices. Since we could not apply rule 4, it must be the case at all vertices in C have neighbors that are not in C. Then each such vertex must be an element of another unique clique. But there are more than k vertices in C, so this would imply that there are more than k cliques in S, which is a contradiction. Then every element of S has at most k vertices.

There are at most k cliques in S, and each has at most k vertices. Since rule 1 could not be applied, every vertex must be in at least one clique. Then there are at most k^2 vertices in the fully reduced graph G.

Once we have a reduced graph bounded in size by k^2, we can use any suitable search algorithm to determine whether the reduced graph is a yes-instance. Remember that this step does not need to be polynomial in k, but merely computable. The paper we read presented an algorithm that ran in $O(2^{k^3}n)$, which is exponential in k but linear in n.

References

[1] Falk Hffner, Rolf Niedermeier, and Sebastian Wernicke. Techniques for practical fixed-parameter algorithms The Computer Journa, 51(1):725, (2008).

[2] CHEKURI, C., and MOTVANI, R. Precedence constrained scheduling to minimize sum of weighted completion times on a single machine Discrete Applied Mathematics. 29-38 (1999).

[3] E. Mujuni and F. Rosamond. Parameterized Complexity of the Clique Partition Problem. In the fourteenth Computing: The Australasian Theory Symposium, 2008.