SEMINAR ON EXACT EXPONENTIAL ALGORITHMS – DYNAMIC PROGRAMMING

JUHO-KUSTAA KANGAS

1. Introduction

A very general approach to solving a complex problem is to break it down into simpler subproblems such that the solution can be efficiently composed from subproblem solutions. Typically subproblems are further broken down up until a trivial case, yielding a recurrence often expressed in a recursive form. Techniques such as branching evaluate such recurrences directly, which can be very efficient if the subproblem composition recursive in nature. When this is not the case, a different method might yield a better solution.

Dynamic programming is a common method for solving a recurrence with *overlapping* subproblems, where a single subproblem solution contributes to those of several larger subproblems. Instead of recursive evaluation, dynamic programming takes a bottom-up approach, solving the simplest subproblems first, then iteratively composing more complex solutions from those already computed. This requires storing each solution in memory, sometimes called *memoization*, and often leading to an exponential space complexity.

Contrary to branching, dynamic programming is useful for designing both polynomial and non-polynomial time algorithms. A well-known example of the former is the Floyd-Warshall algorithm, which computes the shortest paths between all vertices in a graph. In this paper we shall focus on non-polynomial cases, where dynamic programming is typically used to solve problems with a superexponential search space in exponential time.

The contents of this paper are mainly based on the chapters 1, 3 and 10 of Exact Exponential Algorithms (Fomin & Kratch, 2010). In section 2 we present the classical dynamic algorithm for the travelling salesman problem. In sections 3 and 4 we discuss solving TSP more efficiently in terms of time and space, respectively. In section 5 we present the dynamic approach to graph coloring and in section 6 we conclude with some exercises.

Date: 18 October, 2011.

2. Travelling salesman problem

Given a city network and the distances between all pairs of cities, the travelling salesman problem (TSP) asks for a tour of shortest length that visits each city exactly once and returns to the starting city. Formally, given a (usually undirected) graph (V, E) of n vertices and for each pair of vertices $(u, v) \in E$ a related cost $d(u, v)$, the task is to find a bijection $\pi : \{1, \cdots, n\} \to V$ such that the sum

$$\left(\sum_{i=1}^{n-1} d(\pi(i), \pi(i+1))\right) + d(\pi(n), \pi(1))$$

is minimized.

The TSP is a permutation problem; it asks for a permutation π of vertices that minimizes the sum of edges costs between adjacent vertices. The trivial solution is to evaluate the sum for all $n!$ permutations, leading to $\mathcal{O}^*(n!)$ running time, asymptotically worse than any exponential complexity. The TSP is known to be NP-hard while the decision variant, determining whether a tour under given length exist, is NP-complete. Thus, it is generally believed that a polynomial time algorithm does not exist. However, we can still do considerably better than the brute-force solution.

The fastest known algorithm for TSP was discovered independently by Bellman and Held & Karp already in the 1960s. A classical example of dynamic programming, it solves the problem in time $\mathcal{O}^*(2^n)$ by computing optimal tours for the subsets of the vertex set. Specifically, given an arbitrary starting vertex s, for every nonempty $U \subset V$ and every $e \in U$ we compute the length of the shortest tour that starts in s, visits each vertex in U exactly once and ends in e. Denote the length of such tour by $OPT[U, e]$. All values of OPT are stored in memory.

To solve the problem efficiently, we compute the subproblem solutions in the order of increasing cardinality of U. For $|U| = 1$ the task is trivial: the length of the shortest tour starting in s and visiting the single $e \in U$ is simply $d(s, e)$. To obtain the shortest tour for $|U| > 1$, we consider all vertices $u \in U \setminus \{e\}$ for which the edge (u, e) may conclude the tour. If a tour containing (u, e) is optimal, then necessarily the subtour on $U \setminus \{e\}$ ending in u is optimal as well. Since optimal tours on $U \setminus \{e\}$ have already been computed, we need only minimize over all u:

$$d(U, e) = \min_{u \in U \setminus \{e\}} OPT[U \setminus \{e\}, u] + d(u, e).$$

Finally, the value of $OPT[V, s]$, the length of the shortest tour starting and ending in s and visiting all vertices, is the solution of the entire problem. The number of subproblem solutions computed is $\mathcal{O}(2^n n)$. For each of them, the evaluation of the recurrence runs in $\mathcal{O}(n)$ time. Thus, the algorithm runs within a polynomial factor of 2^n. Although exponential, it is a significant improvement over the factorial running time of the brute-force solution.

3. TSP in bounded degree graphs

Despite fifty years of research and progress on other NP-hard problems, the general case exact TSP has seen no improvement since the discovery of the dynamic algorithm. It remains an open question whether the problem can be solved in time $\mathcal{O}^*((2-\epsilon)^n)$ for some $\epsilon > 0$. So far such better bounds have been proven for certain restricted graph classes such as graphs where the maximum degree of each vertex, Δ, is bounded. A branching algorithm by Eppstein solves the problem in time $\mathcal{O}^*(1.260^n)$ for $\Delta = 3$, and in $\mathcal{O}^*(1.890^n)$ for $\Delta = 4$. The former bound was later improved by Iwama and Nakashima to $\mathcal{O}^*(1.251^n)$ and the latter by Gebauer to $\mathcal{O}^*(1.733^n)$.

In this section we present a recent improvement by Björklund & al. over the $\mathcal{O}^*(2^n)$ running time for all Δ. The basic observation, common in dynamic algorithms running over subsets, is that some special subsets can be safely ignored. Specifically, the recurrence needs to be evaluated only for subsets S, which induce a connected subgraph i.e. all vertices in S are connected by a path within S. Whether a subset is connected can be checked in $\mathcal{O}(n)$ time. Thus, by modifying the dynamic algorithm to omit non-connected sets, we obtain the running time $\mathcal{O}^*(|\mathcal{C}|)$ where $\mathcal{C}$ is the family of connected sets of the graph.

We now show that for an n-vertex graph of maximum degree Δ, the size of $\mathcal{C}$ is at most $(2^{\Delta+1}-1)^{n/(\Delta+1)} + n = \mathcal{O}((2-\epsilon)^n)$. The key result used in the proof is a combinatorial consequence of Shearer's inequality, stating the following:

Lemma 1. (Shearer) *Let V be a finite set with subsets $A_1, A_2, \cdots, A_k$ such that every $v \in V$ occurs in at least δ subsets. Let $\mathcal{F}$ be a family of subsets of V. For each $1 \le i \le k$ define the projections $\mathcal{F}_i = \{F \cap A_i : F \in \mathcal{F}\}$. Then,*

$$|\mathcal{F}|^\delta \le \prod_{i=1}^{k} |\mathcal{F}_i|.$$

For each $v \in V$ we define the subset $A_v \subset V$ as the closed neighborhood $N[v] = \{u \in V : (u,v) \in E\} \cup \{v\}$. In an ideal case the graph is Δ-regular. If not, we alter the definitions of A_v a bit by adding each vertex $u \in V$ with $d(u) < \Delta$ to $\Delta - d(u)$ sets A_v, chosen arbitrarily. This ensures that each vertex is contained in exactly $\Delta + 1$ sets, and that

$$\sum_{v \in V} |A_v| = n(\Delta + 1).$$

Now consider the projections $\mathcal{C}_v = \{C \cap A_v : C \in \mathcal{C}'\}$ where $\mathcal{C}'$ is the set of connected sets excluding all singletons (subsets with only one vertex). Observe that for each $v \in V$ the set $\{v\}$ is not contained in $\mathcal{C}_v$. This is because $N[v] \subset A_v$ and $C \cap N[v] \neq \{v\}$ unless C is $\{v\}$ or not connected. Thus $|C_v|$ is at most $2^{|A_v|} - 1$. Now, applying this with lemma 1 we get

$$|\mathcal{C}'|^{\Delta+1} \leq \prod_{v \in V} |\mathcal{C}_v| \leq \prod_{v \in V} (2^{|A_v|} - 1).$$

Define $f(x) = \log(2^x - 1)$. Since f is convex, Jensen's inequality gives us

$$\frac{1}{n} \sum_{v \in V} f(|A_v|) \leq f(\frac{1}{n} \sum_{v \in V} |A_v|) = f(\Delta + 1).$$

By taking exponentials and the nth power on both sides we have

$$\prod_{v \in V} (2^{|A_v|} - 1) \leq (2^{\Delta+1} - 1)^n.$$

Finally, adding in the n singletons excluded from $\mathcal{C}'$ and combining the results above, we have

$$|\mathcal{C}| = |\mathcal{C}'| + n \leq (2^{\Delta+1} - 1)^{n/(\Delta+1)} + n$$

concluding the proof.

The idea of considering only connected sets is presented by Björklund & al. as a basic motivation for analyzing TSP in bounded degree graphs. A more thorough analysis by the authors is based on so called *transient sets* of the graph. The basic idea is that, in addition to sets that are not connected, we can also skip sets that would encircle a vertex so that it may never be visited (or possibly visited but never exited). This analysis yields a tighter bound, $\mathcal{O}^*((2^{\Delta+1} - 2\Delta - 2)^{n/(\Delta+1)})$, which, for $\Delta \geq 5$, is the best known to date.

4. TSP in polynomial space

In practical applications the greatest restriction with dynamic algorithms is often not the time but the exponential amount of space required. In the case of TSP we need to store at least the previous level of subproblems, which is $\mathcal{O}^*(2^n)$ in size. If possible, we might prefer to solve the problem in less space, even if doing so makes the algorithm somewhat slower. While the brute-force search accomplishes exactly this, far better tradeoffs exist.

We shall briefly describe an algorithm for solving TSP in polynomial space and $\mathcal{O}^*(4^n n^{\log n})$ time. Assume for simplicity that n is a power of 2. The key idea is to select the set of the first $\frac{n}{2}$ vertices in the tour, which can be done in 2^n ways. For all possible selections we solve the problem recursively.

For all nonempty $U \subset V$ and $s, e \in U$, define $OPT[U, s, e]$ as the length of the shortest tour that starts in s, visits all vertices in U and ends in e. Trivially, we set $OPT[U, s, e] = d(s, e)$ for $|U| = 2$. Now consider a tour T for $|U| \geq 3$. Let U_1 be the set of the first half of the vertices visited and $U_2 = U \setminus U_1$. Let x be the last vertex visited in U_1 and y the first vertex visited in U_2. If T is optimal, necessarily the subtours on U_1 and U_2 are optimal as well. Thus, $OPT[U, s, e]$ is the minimum of

$$OPT[U_1, s, x] + d(x, y) + OPT[U \setminus U_1, y, e]$$

over all $U_1 \subset U$ of size $|U|/2$ where $s \in U_1$ and $e \notin U_2$. Optimal subtours are computed recursively. The length of the shortest tour in the graph is

$$\min_{s,e \in V; s \neq e} OPT[V, s, e] + d(e, s).$$

We omit the analysis. A hybrid algorithm that switches from recursion to dynamic programming for sufficiently small subproblems achieves a more balanced tradeoff, running in $\mathcal{O}^*(2^{n(2-1/2^i)} n^i)$ time and $\mathcal{O}^*(2^{n/2^i})$ space for any parameter $i \in \{0, 1, 2, \cdots, \lceil \log_2 n \rceil\}$.

5. Graph coloring

A k-coloring of an undirected graph $G = (V, E)$ assigns one of $k \in \mathbb{N}$ distinct colors to each vertex such that adjacent vertices have different colors. For $k \geq 3$ deciding whether G has a k-coloring is NP-complete. Finding smallest such k, called the *chromatic number* of G and denoted by $\chi(G)$, is NP-hard. A coloring using exactly $\chi(G)$ colors is called an *optimal coloring*. Depending on formulation, the graph coloring problem asks for either $\chi(G)$ or an optimal coloring of G.

Graph coloring is an example of a partition problem: it asks for a partition of vertices into color classes. For a n-vertex graph, the total number of partitions is n^n. A trivial brute-force solution enumerates all of them, thus running in time $\mathcal{O}^*(n^n)$. Again, we can reduce the running time to exponential complexity by solving the problem with dynamic programming over the induced subgraphs of G.

First recall some definitions. We say that a subset $I \subset V$ of vertices is an *independent set* if I contains no adjacent vertices. An independent set I is *maximal* if no proper superset of I is independent. Denote by $\mathcal{I}[G]$ the family of **maximal** independent sets of G. We observe that a k-coloring is actually a partition of V into k independent sets. Also, any k-coloring can be modified so that one of the independent sets is maximal. Thus, there is always an optimal coloring with a maximal independent set.

For each $U \subset V$ we compute $OPT[U] = \chi(G[U])$, the chromatic number of the subgraph induced by U. Again, we do the computation in the order of increasing cardinality of U, setting first $OPT[\emptyset] = 0$. For $|U| > 0$, we consider all maximal independent sets I of $G[U]$. We know that for some I an optimal coloring for $G[U]$ consists of I and an optimal coloring for $G[U \setminus I]$. Since the values $OPT[U \setminus I]$ have already been computed, we can minimize over all such colorings:

$$OPT[U] = 1 + \min_{I \in \mathcal{I}[G[U]]} OPT[U \setminus I].$$

Finally, $OPT[V]$ is the solution for the entire graph. The only non-trivial part remaining is the enumeration of maximal independent sets. A brute-force method enumerates all subsets one by one and checks which are maximally independent. Although this is enough to achieve an exponential time, it turns out the enumeration can be done within a polynomial factor of $|\mathcal{I}|$, which for a subgraph of i vertices is at most $3^{i/3}$. Thus, the total number of steps is within a polynomial factor of

$$\sum_{U \subset V, U \neq \emptyset} |\mathcal{I}(G[U])| \leq \sum_{i=1}^{n} \binom{n}{i} \cdot 3^{i/3} \leq (1 + \sqrt[3]{3})^n < 2.4423^n$$

leading to $\mathcal{O}^*(2.4423^n)$ running time. By using a combination of dynamic programming and another technique, inclusion-exclusion, the bound can be further improved even to $\mathcal{O}^*(2^n)$.

6. Conclusion

Dynamic programming is a method for solving a problem by breaking it down into subproblems. Each subproblem solution is stored in memory and the problems are solved from simpler to more complex, composing more complex solutions from the ones already computed. This is often much faster than a naive solution if the subproblems are suitably overlapping and only slightly smaller than the problem they compose. In exponential time algorithms subproblems are often defined in terms of subsets of the original problem, as seen in TSP and graph coloring.

For most of the material in this paper we refer to Fomin & Kratch [2]. For the travelling salesman problem in bounded degree graphs and the account of earlier results see [1].

Exercise 2. (F&K 3.2.) For an undirected graph $G = (V, E)$, the cutwidth of a permutation $\pi : V \to \{1, \cdots, n\}$ is

$$\max_{v \in V} |\{(u, w) \in E : \pi(u) \leq \pi(v) \leq \pi(x)\}|.$$

The cutwidth of G is the minimum cutwidth taken over all permutations on its vertices. Prove that the cutwidth of a graph on n vertices can be computed in time $\mathcal{O}^*(2^n)$.

Exercise 3. (F&K 3.9.) Given an undirected graph $G = (V, E)$, a subset of vertices $D \subset V$ is called a *dominating set* of G if each $v \in V$ either is in D or has a neighbor in D. The domatic number of G is the largest $k \in \mathbb{N}$ such that there is a partition of V into disjoint sets $V_1 \cup V_2 \cup \cdots \cup V_k = V$ and each V_i is a dominating set of G. Show how to compute the domatic number of an n-vertex graph in time $\mathcal{O}^*(3^n)$.

References

1. Andreas Björklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto, *The travelling salesman problem in bounded degree graphs*, Proceedings of the 35th international colloquium on Automata, Languages and Programming, Part I (Berlin, Heidelberg), Springer-Verlag, 2008, pp. 198–209.
2. F.V. Fomin and D. Kratsch, *Exact exponential algorithms*, Texts in Theoretical Computer Science. an EATCS Series, Springer, 2010.