RWTH Aachen **WS 2010**
Lehrgebiet Theoretische Informatik Exam
Kneis—Langer—Rossmanith—Sikdar 14.2.2011

Exact Algorithms

Here we give fairly detailed solutions to the problems in the first exam.

Solution to Exercise 1

A simple recursive algorithm may essentially branch as follows: Given a set $C \subseteq U$, select any uncovered tuple $v = (x_1, x_2, x_3) \in V$. To make C cover v, we clearly need $x_1 \in C$ or $\{x_2, x_3\} \subseteq C$. Thus it suffices to test whether there is a minimal solution with $x_1 \in C$, or whether there is a minimal solution with $x_1 \notin C$, but $\{x_2, x_3\} \subseteq C$. Then we just need to add a few straightforward reduction rules, for instance in order to handle cases where x_1 was already fixed as $x_1 \notin C$ or either of $x_2, x_3 \notin C$.

A concrete algorithm may work as follows. It is given the instance (U, V) and two additional sets $C, \bar{C} \subseteq U$. Here C is the set that we want to "extend" to a solution, while $\bar{C}$ is the complement of C, i.e., the set of elements that we fixed as not being members of C.

Algorithm COVER$(U, V, C, \bar{C})$

1. If C covers every $v \in V$, then return $|C|$.
2. If $U = C \cup \bar{C}$, then return ∞.
3. Choose any $v = (x_1, x_2, x_3) \in V$ not covered by C.
4. If $x_1 \in \bar{C}$:
   (a) If $\{x_2, x_3\} \cap \bar{C} \neq \emptyset$ then return ∞
   (b) return COVER$(U, V, C \cup \{x_2, x_3\}, \bar{C})$
5. If $\{x_2, x_3\} \cap \bar{C} \neq \emptyset$:
   (a) If $x_1 \in \bar{C}$ then return ∞
   (b) return COVER$(U, V, C \cup \{x_1\}, \bar{C})$
6. Otherwise return $\min\big\{\text{COVER}(U, V, C \cup \{x_1\}, \bar{C}), \text{COVER}(U, V, C \cup \{x_2, x_3\}, \bar{C} \cup \{x_1\})\big\}$.

We briefly explain the algorithm. In the first line, we check whether the given C is a solution; if so, we can return its size. In the second line, we check whether there are any unset elements left, i.e., elements $u \in U$ for which we still need to decide whether $u \in C$ or $u \notin C$ (in other words, $u \in \bar{C}$. If this isn't the case, the set C is no solution and we return ∞. Then we choose any uncovered $v = (x_1, x_2, x_3) \in V$. Such a tuple must exist since C is not a solution. We now apply two straightforward reduction rules: If either of $x_1, x_2, x_3 \in \bar{C}$, then the only way to cover v is to choose the other elements. For instance, if $x_1 \in \bar{C}$, then v can only by covered by adding both, x_2 and x_3 to C. This is only possible if neither of x_2 or x_3 is already a member of $\bar{C}$. Therefore, we either return ∞, if the requirement is violated, or we do not need to branch at all and call the algorithm recursively. Finally we branch on the two main cases as explained above.

The recursive search tree of the algorithm has height bounded by $n = |U|$ since in each step we add at least one variable to either C or $\bar{C}$, and hence after at most recursive n calls we have $|U| - |C| - |\bar{C}| = 0$. Its branching vectors are as follows:

- If either of $x_1, x_2, x_3 \in \bar{C}$, we do not branch.

- If $x_1, x_2, x_3 \in U \setminus (C \cup \bar{C})$, then
$$(|U| - |C| - |\bar{C}|) - (|U| - |C \cup \{x_1\}| - |\bar{C}|) = 1$$
and
$$(|U| - |C| - |\bar{C}|) - (|U| - |C \cup \{x_2, x_3\}| - |\bar{C} \cup \{x_1\}|) = 3,$$
i.e., a branching vector of $(1, 3)$.
- If $x_2 \in C$ and $x_1, x_3 \in U \setminus (C \cup \bar{C})$, then
$$(|U| - |C| - |\bar{C}|) - (|U| - |C \cup \{x_1\}| - |\bar{C}|) = 1$$
and
$$(|U| - |C| - |\bar{C}|) - (|U| - |C \cup \{x_2, x_3\}| - |\bar{C} \cup \{x_1\}|) = 2,$$
i.e., a branching vector of $(1, 2)$.
- If $x_3 \in C$ and $x_1, x_2 \in U \setminus (C \cup \bar{C})$, then we also get a branching vector of $(1, 2)$.

Here, $(1, 2)$ is the worst branching vector. Therefore, the algorithm's running time is bounded by $O^*(\tau(1, 2)^n)$.

Solution to Exercise 2

Let $n = |V|$ and V_1, V_2, V_3 be a three-partition of V such that $\lfloor n/3 \rfloor \leq |V_i| \leq \lceil n/3 \rceil$ for each $1 \leq i \leq 3$. For the ease of presentation we let $n = 3k$ for $k \in \mathbf{N}$. Let for $1 \leq i \leq 3$ be $V_i' = \{ U_i \subseteq V_i \}$. Then clearly $|V_i'| = 2^{n/3}$. The algorithm constructs the graph $G' = (V', E')$ with $V' = V_1' \cup V_2' \cup V_3'$ and
$$E' = \bigcup_{1 \leq i \leq 3} \{ (u, v) \in V_i' \times V_{(i+1 \bmod 3)+1}' \mid u \cup v \text{ is independent in } G \}.$$
For $e = (u, v) \in E'$ we then let $w(e) := |u|$. Then $|V'| = 3 \cdot 2^{n/3} \in O(2^{n/3})$ and $|E'| \in O((2^{n/3})^2)$, and by definition for each triangle $u_1, e_1, u_2, e_2, u_3, e_3, u_1$ with edges e_1, e_2, e_3 and vertices u_1, u_2, u_3 in G' the set $u_1 \cup u_2 \cup u_3$ is an indenpendent set in G and $w(e_1) + w(e_2) + w(e_3) = |u_1| + |u_2| + |u_3| = |u_1 \cup u_2 \cup u_3|$. Conversely, let $I \subseteq V$ be an independent set of G. Then for each $1 \leq i \leq 3$ there is a node $u_i = I \cap V_i \in V_i'$, and there are edges $e_1 = (u_1, u_2)$, $e_2 = (u_2, u_3)$ and $e_3 = (u_3, u_1)$ in E'. The triangle with edges e_1, e_2, e_3 then has weight $w(e_1) + w(e_2) + w(e_3) = |u_1| + |u_2| + |u_3| = |I|$.

We can then solve the Independent Set problem as follows: Let for k_1, k_2, k_3 be G'_{k_1,k_2,k_3} the subgraph of G' such that edges from V_1' to V_2' have weight k_1, edges from V_2' to V_3' have weight k_2 and edges from V_3' to V_1' have weight k_3. Let for k_1, k_2, k_3 be $A[k_1, k_2, k_3]$ be the adjacency matrix of G'_{k_1,k_2,k_3} of order $3 \cdot 2^{n/3} \times 3 \cdot 2^{n/3}$. Clearly, $A[k_1, k_2, k_3]$ can be computed in time polynomial in $2^{n/3}$. Furthermore, $A[k_1, k_2, k_3] \times A[k_1, k_2, k_3] \times A[k_1, k_2, k_3]$ has a positive entry iff there is a triangle in G'_{k_1,k_2,k_3}. By construction, this triangle then has weight $k_1 + k_2 + k_3$. We therefore can use two matrix multiplications that require time $O^*(2^{\omega n/3})$ each to find triangles in G'_{k_1,k_2,k_3}. To find the maximum independent set, we need to do this for all $0 \leq k_1, k_2, k_3 \leq n$.

Solution to Exercise 3

Given a graph $G = (V, E)$, for $S \subseteq V$ define OPT$[S]$ as follows:
$$\text{OPT}[S] := \text{domatic number of the graph } G[S].$$
For $S = \emptyset$, clearly OPT$[S] = 0$. For $\emptyset \neq S \subseteq V$,
$$\text{OPT}[S] = \max_{X \subseteq S} \{\text{OPT}[G \setminus X]\} + 1,$$
where the maximum is taken over all $X \subseteq S$ such that X is a dominating set of $G[S]$. For a fixed S with i vertices, one can compute the set of all dominating sets of $G[S]$ by enumerating all vertex subsets in time $O^*(2^i)$. Hence the time taken to compute OPT$[S]$ for a fixed S is $O^*(2^i)$ and the total time taken is
$$O^* \left(\sum_{i=0}^{n} \binom{n}{i} \cdot 2^i \right) = O^*(3^n).$$

Solution to Exercise 4

There was a small error in this question. The running time of the algorithm should have been $O^*(2^n \cdot k)$ and *not* $O^*(2^n \cdot \log k)$ as stated in the test. However marks were awarded to anyone who had written down the basic Inclusion-Exclusion machinery used for this problem.

To apply the Inclusion-Exclusion machinery, we need to first define what our objects are and what properties they must satisfy. We define our objects to be *closed* walks that start and end at vertex s, of length $n+1$ and with weight exactly c, where $1 \leq c \leq k$. The objective is to first design an algorithm that tests whether an input graph has a Hamiltonian cycle of weight exactly c. To find out whether there is a Hamiltonian cycle of weight at most k, this algorithm is called k times by setting $c = 1, \ldots, k$ which accounts for the running time of $O^*(2^n \cdot k)$.

An object satisfies property A_i^c for $i \in \{1, \ldots, n\}$ iff it contains the vertex u_i. Therefore the input graph has a Hamiltonian cycle of weight exactly c iff $|\bigcap_{i=1}^n A_i^c| > 0$. By Inclusion-Exclusion,

$$\bigcap_{i=1}^{n} A_i^c = \sum_{X \subseteq [n]} (-1)^{|X|} \left| \bigcap_{j \in X} \bar{A}_j^c \right|.$$

For a fixed $X \subseteq [n]$, $u \in V \setminus X$, $0 \leq p \leq n+1$ and $0 \leq r \leq k$, define $P_X(u, p, r)$ to be the number of walks from s to u in the graph $G[V \setminus X]$ of length p and weight exactly r. The expression $\left|\bigcap_{j \in X} \bar{A}_j^c\right|$ is then given by

$$\left| \bigcap_{j \in X} \bar{A}_j^c \right| = \sum_{u \in N(S) \setminus X} P_X(u, n+1, c).$$

Clearly $P_X(u, 0, r) = 0$ and $P_X(u, p, 0) = 0$. Also define $P_X(u, p, r) = 0$ for all $p, r < 0$. Now

$$P_X(u, p, r) = \sum_{v \in N(u) \setminus X} P_X(v, p-1, r - w(u, v)),$$

where $w(u, v)$ is the weight of the edge (u, v). The time taken for computing $P_X(u, p, r)$ for all u, p, r is $O(mnc)$. The space required is $O^*(c)$. The time taken to compute the $P_X(u, p, r)$ terms for all $X \subseteq \{u_1, \ldots, u_n\}$ is $O(2^n \cdot mnc)$ and the space required is $O^*(c)$. This algorithm is called for $c = 1, \ldots, k$ and hence the total time taken is $O(2^n \cdot k \cdot mnk)$ and the space required is $O^*(k)$.

Exercise 5

1. To show that $(f * g) * h = f * (g * h)$, we need to show that for all $S \subseteq U$

$$((f * g) * h)(S) = (f * (g * h))(S).$$

Fix $S \subseteq U$. Then

$$\begin{aligned}
((f * g) * h)(S) &= \sum_{S_1 \uplus S_2 = S} (f * g)(S_1) \cdot h(S_2) \\
&= \sum_{S_1 \uplus S_2 = S} \left(\sum_{X \uplus Y = S_1} f(X) \cdot g(Y) \right) \cdot h(S_2) \\
&= \sum_{X \uplus Y \uplus Z = S} f(X) g(Y) h(Z),
\end{aligned}$$

where the last sum is over all tri-partitions $X \uplus Y \uplus Z$ of the set S. The last expression on the right hand side may be written as:

$$\begin{aligned}
\sum_{X \uplus Y \uplus Z = S} f(X) g(Y) h(Z) &= \sum_{X \uplus S' = S} f(X) \cdot \left(\sum_{Y \uplus Z = S'} g(Y) \cdot h(Z) \right) \\
&= \sum_{X \uplus S' = S} f(X) \cdot (g * h)(S') \\
&= (f * (g * h))(S).
\end{aligned}$$

This proves that the convolution product is indeed associative.

2. We have to show that $\mu(\zeta(f))(S) = f(S)$ for all $S \subseteq U$. For a fixed $S \subseteq U$,

$$\begin{aligned}\mu(\zeta(f))(S) &= \sum_{X\subseteq S}(-1)^{|S\setminus X|}\zeta(f)(X)\\ &= \sum_{X\subseteq S}(-1)^{|S\setminus X|}\sum_{Y\subseteq X} f(Y)\\ &= \sum_{Y\subseteq S}\left(\sum_{X\supseteq Y, X\subseteq S}(-1)^{|S\setminus X|}\right)\cdot f(Y).\end{aligned}$$

Note the change in the order of summation. The outer sum is over all subsets Y of S while the inner sum is over all subsets $X \subseteq S$ that contain Y. Now there are two situations to consider: when $Y \subset S$ and when $Y = S$. When $Y = S$, then $Y = X = S$ and the inner sum equals 1. When $Y \subset S$ then the coefficient of $f(Y)$ depends on the number of supersets $X \supseteq Y$. Let $|S| = s$ and $|Y| = p$. If $|X| = p + i$, where $0 \leq i \leq s - p$, then there are $\binom{s-p}{i}$ choices for the set X and for each choice of X, the contribution to the coefficient of $f(Y)$ is $(-1)^{s-p-i}$. Thus the coefficient of $f(Y)$ is

$$\sum_{i=0}^{s-p}\binom{s-p}{i}(-1)^{s-p-i} = (-1)^{s-p}\sum_{i=0}^{s-p}\binom{s-p}{i}(-1)^{i} = (-1)^{s-p}(1-1)^{s-p} = 0.$$

3. Define $f\colon 2^V \to \{0,1\}$ as follows: for $S \subseteq V$,

$$f(S) := \begin{cases} 1, & \text{if } S \text{ is independent in } G;\\ 0, & \text{otherwise.}\end{cases}$$

Induction Basis. For $k = 2$, the convolution product $(f * f)(V)$, by definition, is

$$(f * f)(V) = \sum_{X\uplus Y = V} f(X)\cdot f(Y).$$

The right-hand side is strictly positive iff $f(X) > 0$ and $f(Y) > 0$ for some bipartition $X \uplus Y$ of V where both X and Y are independent. Hence the right-hand side is strictly positive iff the graph has a two-coloring.

Induction Hypothesis. Assume that $(f * \cdots * f)_k(V) > 0$ iff the graph G has a k-coloring.

Induction Step. The expression $(f * \cdots * f)_{k+1}(V)$ may be written as:

$$\begin{aligned}(f * \cdots * f)_{k+1}(V) &= ((f * \cdots * f)_k * f)(V)\\ &= \sum_{X\uplus Y = V}(f * \cdots * f)_k(X)\cdot f(Y).\end{aligned}$$

Now the right-hand side is strictly positive iff there exists a bipartition $X \uplus Y$ of V such that $(f * \cdots * f)_k(X) > 0$ and $f(Y) > 0$. This holds iff the graph $G[X]$ is k-colorable (by induction hypothesis) and Y is independent in G which holds iff the graph is $(k+1)$-colorable.