Problem set 7

Jonas Ishøj Nielsen, join@itu.dk

01/12/2021

Contents

1 Inclusion-exclusion algorithms

1.1 Inclusion-exclusion count perfect matchings

The code can be found in `InclusionExclusion`$_{C}ountPerfectMatchings.py$.

1.2 Test the limit

On a square grid of size 8x2 then it takes around 4 seconds and a 9x2 graph will take around 15 seconds.

1.3 Inclusion-exclusion count Hamiltonian cycles

The code can be found in CountHamiltonianCycles.py. Timewise for the inclusion-exclusion it can compute on a graph with 16 vertices in reasonable time, however at around 13 vertices it begins giving "overflow encountered in long_scalars" when calling "np.linalg.matrix_power". The DP is slightly faster and can compute a graph with 17 around as fast as the inclusion-exlusion can count one with 16 vertices, and it doesn't give any overflows.

1.4 Inclusion-exclusion on TSP

Idea is to compute:

$$count = \sum_{S\subset\{1,\dots,n\}} (-1)^{|S|} \cdot |\wedge_{i\in S} B_i| = \sum_{S\subset\{1,\dots,n\}} (-1)^{|S|} \cdot \Delta_S$$

where Δ_S=the sum of the diagonals on the augmented matrix of G' without S to the power of n. In this, G' is the values of G where edge weight $w_{u,v}$ with $x^{w_{u,v}}$ and is thus a polynomial. Δ_S basically ends up representing the possible walks of length n that doesn't use any edges in S.
The final result is then: the index of the smallest coefficient in the result.

2 Reduce to SAT

Functions

For ease of describing, 3 functions have been created to determine how many of the variables have the same value as the literals.
At most k:

$$<=_k (x_1, x_2, ..., x_t) := \forall_{1<=j_1<j_2<....<j_{k+1}<=t}(\bar{x_{j_1}} \vee \bar{x_{j_2}} \vee ... \vee \bar{x_{j_{k+1}}})$$

At least k:

$$>=_k (x_1, x_2, ..., x_t) := \forall_{1<=j_1<j_2<....<j_{t-k+1}<=t}(x_{j_1} \vee x_{j_2} \vee ... \vee x_{j_{j-k+1}})$$

Exactly k:

$$==_k (x_1, x_2, ..., x_t) :=<=_k (x_1, x_2, ..., x_t) \wedge >=_k (x_1, x_2, ..., x_t)$$

2.1 Vertex cover, input: graph G and k

Add one variable x_v for each v ∈ V.
For each edge (u,v) ∈ E: add clause $(x_u \vee x_v)$
Add the clauses $==_k (x_1, x_2, ..., x_n)$

2.2 k-coloring, input: graph G and k

Add one variable $x_{v,c}$ for each pair v,c with v ∈ V and c $\in \{1, ..., k\}$.
For each pair v,c: add the clauses: $==_1 (x_{v,c}, x_{n_1,c}, ..., x_{n_t,c})$ with nodes $n_1, ..., n_t$ being the neighbors of v.
For each v ∈ V: add the clauses $==_1 (x_{v,1}, x_{v,2}, ..., x_{v,k})$

2.3 Hamiltonian cycle, input: graph G

Add one variable $x_{u,v}$ for each (u,v) ∈ E.
For each v ∈ V: add the clauses $==_2 (x_{v,n_1}, x_{v,n_2}, ..., x_{v,n_t})$ with nodes $n_1, ..., n_t$ being the neighbors of v.
Add the clauses $==_n (x_1, ..., x_m)$ with $x_1, ..., x_m$ being all the variables

2.4 Implementation hamiltonian cycle

The implementation can be found in SATReduction.py.

3 LP maximum weight matching

The code can be found in LP$_maximum_weight_matching.py.ItwillworkinvastmajorityofcasesHOWEVER:$ $thereferencedLPsolverhaserrorsinitsboundsandthusthevariablescanenduphavingavalueabove1, despitetheboundbeing$ the floating point calculations will result in cases where searching for a cycle fails to find one due to an edge having an odd number of non-integral edges. To handle this I rounded that non-integral edge, as in my testing it was so close to 0 or 1 that it was negligible.

4 k-clique and 3-coloring with split-and-list technique

Choose a k s.t. |V|/k >=3.
Split the edges into k partitions.
List all valid 3-colorings for each partition.
Make each coloring a node, and add an edge to each node in another partition if both colorings are valid.
Run clique on the created graph G, and if a clique exists, the selected nodes can be combined to a 3-coloring.

Analysis:
Total number of Nodes: n'=$O(2^{n/k} \cdot k)$
An $O(2^{\epsilon' n}$ for 3-coloring implies an $O(n'^{\epsilon \cdot k}$ for k-clique.

$$O(n'^{\epsilon \cdot k} = O((2^{n/k} \cdot k)^{\epsilon \cdot k}$$

$$= O((2^{n/k})^{\epsilon \cdot k} \cdot k^{\epsilon \cdot k}$$

Since k and are constants:

$$= O(2^{\epsilon \cdot n} \cdot O(k) = O(2^{\epsilon \cdot n}$$

Thus if there exists an $O(n'^{\epsilon \cdot k}$ algorithm for k-clique then there exists an $O(2^{\epsilon' n}$ algorithm for 3-coloring.