MTAT.07.004 — Complexity Theory December 9th, 2011

Lecture 13

Lecturer: Peeter Laud *Scribe(s): Alisa Pankova, Ilya Kuzovkin*

Class #P

Definition 1 Function $f : \{0,1\}^* \to \mathbb{N}$ belongs to class #P if there is a language in NP, or, more precisely, a NTM M', such that $f(x)$ is the number of certificates that $M'(x)$ accepts. More formally, $f \in \#\mathsf{P}$ if there is a DTM M and a polynomial p, such that

$$f(x) = |\{y \in \{0,1\}^{p(|x|)} \mid M(x,y) = true\}|$$

♢

Definition 2 Class FP is class of all function $f : \{0,1\}^* \to \mathbb{N}$ computable in polynomial time. ♢

Theorem 1 *If $FP = \#\mathsf{P}$ then $\mathsf{P} = \mathsf{NP}$.*

Proof We want to show that if $\mathsf{FP} = \#\mathsf{P}$ then $\mathsf{NP} \subseteq \mathsf{P}$. Solving any FP problem takes polynomial time. Our assumption implies that for any NP problem we can learn the number of certificates in polynomial time. Hence, if $L \in \mathsf{NP}$, then we can check whether $x \in L$ by determining the number of acceptance certificates for x of some NTM recognizing L. If the number of certificates is above zero, then $x \in L$. ■

Hardness of finding vs. counting

Let's have a look at CYCLE problem for a directed graph. The question it answers is "Does graph G contain a simple (no repeating vertices or edges, except for starting vertex) cycle?". This problem can be solved in polynomial time: CYCLE $\in$ P.

Next question to ask is how many simple cycles graph G has. This problem is referred as #CYCLE and #CYCLE $\in$ #P.

Theorem 2 *If $\#CYCLE \in FP$ then $P = NP$*

Proof If #CYCLE is in FP, then for any graph we can find the number of cycles in polynomial time. Now assume we have graph G and we want to know if it is Hamiltonian (a NP-complete problem). We introduce new graph G' which is constructed as is shown on Figure 1: for every pair of vertices we add $n \log n$ layers between them, so than there will be $2^{n \log n + 1}$ possible paths from u' to v'.

In G the number of cycles of length at most $n-1$ is upper-bounded by n^{n-1} because we

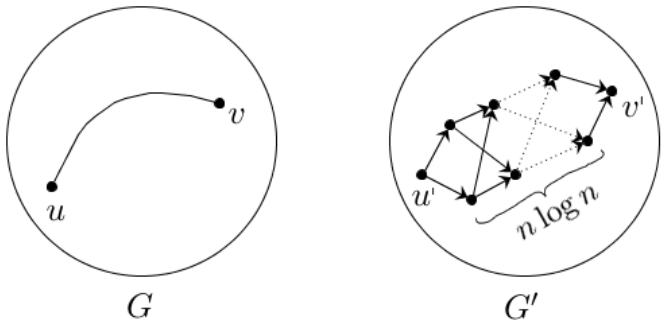

Figure 1: G' size is polynomial of G size

can assign to any cycle a sequence of $n-1$ vertices by starting from some vertex in that cycle and making $n-2$ steps along it; different cycles will give different sequences. For each cycle of length m in G there are

$$2^{m(n\log n+1)}$$

cycles in G'. Hence, if G' is not Hamiltonian (there are no cycles of length n) then the upper bound for the number of cycles in G' is

$$n^{n-1}\cdot 2^{(n-1)(n\log n+1)} = 2^{n(n\log n+1)-\log n-1} \quad .$$

Also we know that if G is Hamiltonian then G' will have at least

$$2^{n(n\log n+1)}$$

cycles.

Problem of determining if graph is Hamiltonian is known to be NP-complete. Using our approach we can check if number of cycles in graph G' is larger or equal than $2^{n(n\log n+1)}$. If it is larger or equal, then G is Hamiltonian, if it is smaller, then it is not Hamiltonian. Since all our computation was made in polynomial time in FP (by assumption) then it turns out that we can solve NP-complete problem in polynomial time, which will imply NP $\subseteq$ P (and P $\subseteq$ NP is obvious).

■

#P-completeness

Definition 3 Function $f : \{0,1\}^* \to \mathbb{N}$ is #P-complete if $f \in \#P$ and $FP^f = \#P$. In other words completeness is achieved when function belongs to the class (obviously) and polynomial functions $g \in$ FP using f-Oracle can solve any problem in #P. ♢

Theorem 3 *#SAT is #P-complete.*

#SAT answers the question of how many satisfying valuations particular Boolean formula has.

Proof The proof goes exactly as for "SAT is NP-complete" theorem from Lecture 3. Only difference is that now we are interested not only in the fact some of the branches will accept, but also in how many of them will accept. We can compute it since our reduction of SAT to a graph preserved all the computation paths and the certificates. ■

Perfect matching in bipartite graphs

Definition 4 Perfect matching of undirected (not necessarily bipartite) graph G is set S of edges, such that each vertex of a graph is incident with exactly one edge in S. ♢

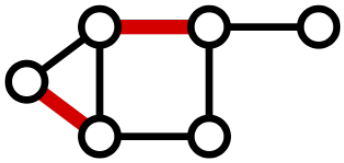

Figure 2: Not a perfect matching

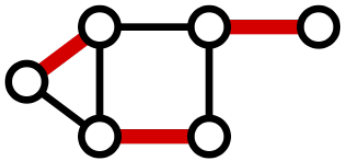

Figure 3: Perfect matching

Definition 5 Graph G is called bipartite is it's vertices set V can be divided into two disjoint subsets V_1 and V_2 such that there is no edge between any two vertices in V_1 and no edge between any two vertices in V_2 (all existing edges go from V_1 to V_2 or vice versa). ♢

We will consider bipartite graphs where $|V_1| = |V_2|$.

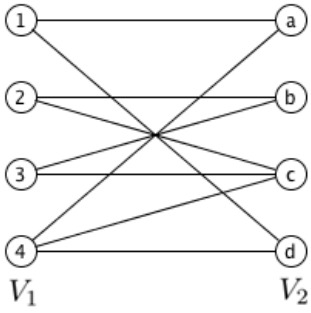

Figure 4: Bipartite graph

Graphs can be represented as adjacency matrices. For a bipartite graph where the sizes of the parts are x and y, the adjacency matrix can have x rows and y columns, where each entry indicates whether there is (1) or is not (0) an edge between the corresponding vertices

in the first and the second part. Here is the representation of the graph from Figure 4.

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

Permanent of a matrix

Definition 6 Permanent of a matrix A is

$$\text{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} a_{i,\sigma(i)}$$

where S_i is a set of a permutations $\{1, \ldots, n\}$.
In other words permanent is sum of multiplications of the elements taken in such way that all factors inside one summand come from different row and column. ♢

$$\text{perm} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} =$$

$$(1 \cdot 1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 0 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 0 \cdot 0) +$$
$$(1 \cdot 0 \cdot 1 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 0) + (0 \cdot 0 \cdot 1 \cdot 1) + (0 \cdot 0 \cdot 0 \cdot 1) +$$
$$(0 \cdot 1 \cdot 0 \cdot 1) + (0 \cdot 1 \cdot 0 \cdot 1) + (0 \cdot 0 \cdot 0 \cdot 1) + (0 \cdot 0 \cdot 1 \cdot 1) +$$
$$(0 \cdot 0 \cdot 1 \cdot 1) + (0 \cdot 0 \cdot 0 \cdot 0) + (0 \cdot 1 \cdot 0 \cdot 1) + (0 \cdot 1 \cdot 0 \cdot 1) +$$
$$(0 \cdot 0 \cdot 0 \cdot 0) + (0 \cdot 0 \cdot 1 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 1) + (1 \cdot 0 \cdot 1 \cdot 0) +$$
$$(1 \cdot 1 \cdot 0 \cdot 1) + (1 \cdot 1 \cdot 1 \cdot 1) + (1 \cdot 1 \cdot 0 \cdot 0) + (1 \cdot 1 \cdot 1 \cdot 1) =$$

$$1+0+1+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+0+1+0+1 = 4$$

Note If summand is 1 then the permutation corresponding to this summand gives us a perfect matching in the graph. If it is 0 then there is no matching. Permanent = number of perfect matchings for a graph.

Definition 7 A *cycle cover* of a directed graph is a set of cycles, such that each node is on exactly one cycle. ♢

Theorem 4 *Perm is #P-complete (even for 0,1-matrices)*

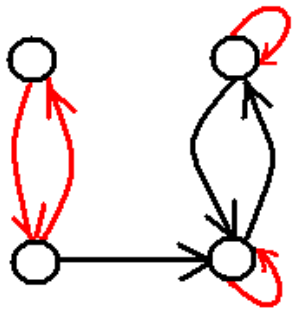

Figure 5: Cycle cover

Proof First of all, note that the permanent of the adjacency matrix of an arbitrary directed graph denotes the number of cycle covers in it. If one summand of the permanent is 1, it means that all the arcs that we have chosen for that summand form a cycle cover. They may belong to a single cycle, but there may also be several cycles. A vertex may be even connected to itself. If the summand is 0, it means that the connection is lost for at least one of the arcs, and it gives us no cycle cover.

If the arcs of a graph have weights, we may include the weights into the adjacency matrix. This permanent would be less sensible (it does not represent the number of cycle covers anymore), but we will need it for our further proof. If the arcs have weights then we define the weight of a cycle cover as the product of the weights of the arcs in this cover. In such manner, the permanent of the adjacency graph would equal the sum of the weights of all cycle covers.

Now we are going to show how to reduce #SAT to *perm*. Since we have already proved that #SAT is $\#P$-complete, we will show that *perm* is also $\#P$-complete. First, we show that it holds for the permanents that use arc weights, and afterwards we show that it holds even for 0,1-matrices.

In a previous lecture, we have seen that 3-CNFSAT is NP complete. Furthermore, the reduction (from SAT) we used to show this preserves the number of certificates. Therefore, it is sufficient to show that we can reduce #3-CNFSAT to *perm*.

Suppose that we are given a boolean formula in 3-$\mathcal{CNF}$.

$$\varphi = C_1 \wedge C_2 \wedge \ldots C_m \ ,$$

where C_i denote clauses that consist of at most three variables:

$$C_i = l_{i1} \vee l_{i2} \vee l_{i3} \ .$$

We have shown that *perm* denotes the number of possible cycle covers in a graph. Now we need to construct a graph G corresponding to the formula φ, such that the number of satisfiable evaluations in φ is equal to the number of cycle covers in G. This construction is not trivial, and first of all it requires construction of some auxiliary parts (so-called gadgets).

First of all, we will construct a graph G'. It will be similar to G, but it will contain some weighted edges. Later, we will show how to get rid of them.

- **Variables:** for each variable in φ, we define the following construction:

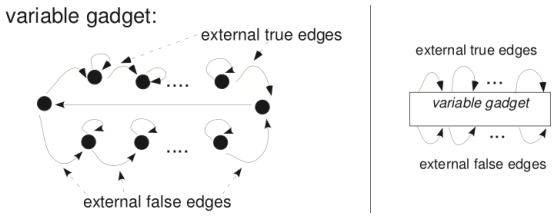

Figure 6: Variable gadget

Each outer edge represents an association with a clause. The numbers of true- and false-edges are equal to the number of times the variable occurs in a clause either positively or negatively. Note that the variable cannot be at once true and false in a single circuit over (see Figure 7).

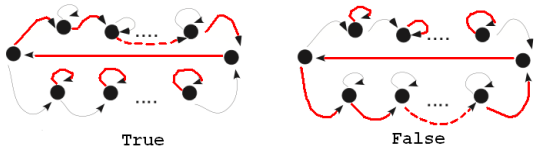

Figure 7: True and False states of a variable

- Clauses: for each clause in φ, we define the following construction:

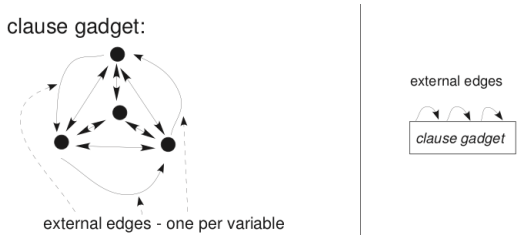

Figure 8: Clause gadget

The external edges here represent associations with variables. It works in a reversed way: if the variable is true, then its corresponding external edge belongs to the cycle cover, but the edge on the clause's side does not belong to the cycle cover. We can represent any true-false combination of the three variables on this graph, as it is shown in Figure 9.

- XOR: unfortunately, we cannot connect variables and clauses directly. It is important to maintain correctess: if the variable is true, then it must indeed be true in each

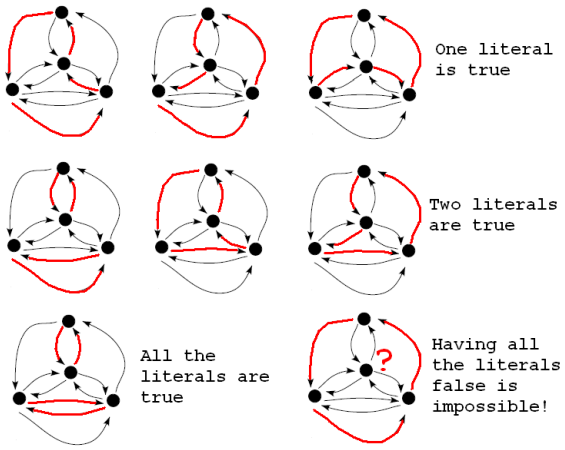

Figure 9: Possible covers of a clause gadget

clause. We need to be sure that exactly one of variable's external edge and the corresonding clause's external edge belong to the cycle cover. They cannot belong to it at once, and it is also impossible that none of them belongs to the cover. We need one more auxiliary construction.

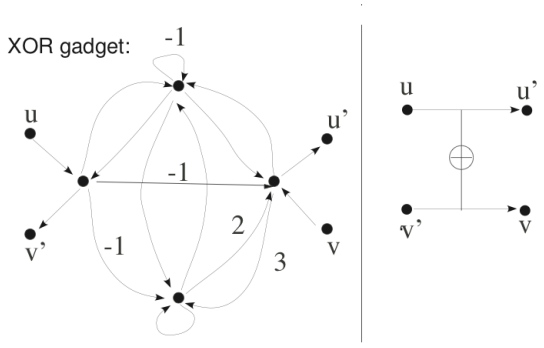

Figure 10: XOR gadget

Let uu' denote an edge in a variable gadget, and vv' in a clause gadget. We can indeed choose edges in such a way that both uu' and vv' do not belong to the cover. Therefore, we need to introduce weights.

In this case we want the *perm* of this construction be 0. Then the *perm* of the whole graph will be 0 (because of multiplication by 0), and this cover will therefore not be included in the answer. The given XOR construction indeed provides this property (see Figure 11).

On the other hand, if we consider exactly one of uu' and vv', we need to be sure that the sum will not be 0. Let us consider all the possibilities for both cases.
As we can see from Figure 12, for the path uu' we have $1+(-1)+2+1+2+(-1) = 4$ cycle covers.

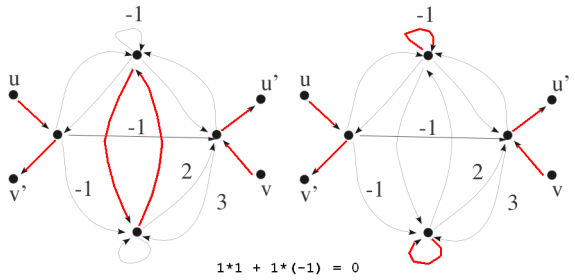

Figure 11: Introducing both uu' and vv'

From Figure 13, for the path vv' we also have $1 + 3 = 4$ cycle covers.

We have got that the number of cycle covers of XOR graph is exactly 4. Since each clause is connected to three variables, there are 3 XOR nodes, and their *perm* together is 4 (neither the variable gadgets nor the clause gadgets do anything to number of cycle covers since their edges have weight 1).

We have m clauses in total. The whole graph G' would have $perm = 4^{3m}$ cycle covers, and that's for only one particular evaluation. In general, there are $\#\varphi$ evaluations that return positive values for the permanent. Therefore, we can say that $perm(G) = 4^{3m} \cdot \#\varphi$. We can compute directly $\#\varphi = perm(G)/4^{3m}$.

We have reduced #SAT to *perm*, but there are some numbers that are neither 0 nor 1. It would be good to reduce the graph G' to the graph G, where all the edges are of weight 1, and where the same properties hold.

- Each positive edge with weight k can be split to k edges of weight 1. The answer does not change since we get k possiblities instead of 1, and summing them together we get back the proper value. The problem is that multiple edges between the same vertices are not permitted. We need to put an intermediate vertex into each of the new edges, and each of these intermediate vertices need a loopback edge that would keep the number of cycle covers consistent.

- There are more problems with the negative edges. We can split them also, but one of them will have weight -1. We want to get rid of negative weights.

  If there are n vertices in a graph (n variables in a formula), then the permanent will be somewhere in the interval $[-n!, n!]$. We can pick m so that $2^m >= 2n!$, and can therefore compute everything *mod* 2^m+1 without loss of precision. The values -1 will therefore become 2^m. We can handle them in the same way as we handled positive edges. In order to reduce exponential number of edges, we may split the edge first of all sequentially. For example, if we have $u \xrightarrow{-1} v$, we first transform it to $u \xrightarrow{2^m} v$, and then to

  $$u \xrightarrow{2} u_1 \xrightarrow{2} \ldots \xrightarrow{2} u_{m-1} \xrightarrow{2} v \ .$$

  We have still some edges left with weight 2. They can be handled as ordinary positive edges (transformed to two edges of weight 1 with additional vertices in the middle).

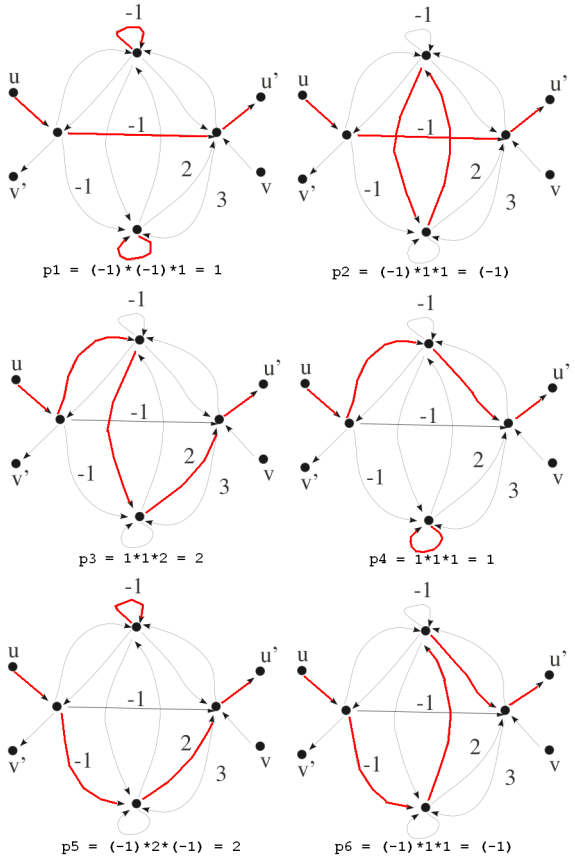

Figure 12: Cycles for uu'

In conclusion, we have reduced $\#SAT$ to *perm*, and since $\#SAT$ solves any problem in $\#P$, we get that *perm* can also solve any problem in $\#P$. We get that *perm* is $\#P$-complete even for 0,1-matrices. ■

Optimization

An *optimization problem* consists of:

- Relation $\rho \subseteq \{0,1\}^* \times \{0,1\}^*$, where $x \ \rho \ y$ means that y is a feasible solution to the problem x.
- *cost function* $c : \{0,1\}^* \to \mathbb{N}$, which computes the cost of an answer to the given problem (each problem uses its own costs).
- Direction of optimization (maximization / minimization).

The optimization problem "does there exist a feasible solution to the problem x of cost at most/least K?" can be denoted as $\langle x, \rho, c, K \rangle$. If ρ, c are polynomial-time functions, then

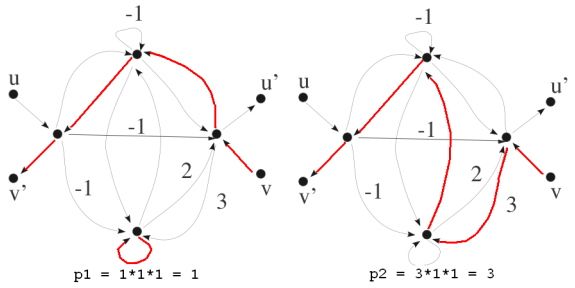

Figure 13: Cycles for vv'

$\langle x, \rho, c, K\rangle \in \mathsf{NP}$. Given a cerificate y, we need to check if $x\ \rho\ y$ and if $c(y) \leq K$. Both things can be done in polynomial time.

An example of an optimization problem is a *maximum satisfiability* problem MAXSAT. Given Boolean formulas $\varphi_1, \ldots, \varphi_m$ with the same variables $x_1, \ldots, x_n$, find a valuation of these variables that satisfies as many of $\varphi_1, \ldots, \varphi_m$ as possible. The answer does not need to satisfy all of them.

A similar problem is k-MAXSAT, where each of the formulas φ_i depends on at most k variables.

There exist more optimization problems: traveling salesman, independent set, vertex cover, knapsack, ...

Approximability

Let $\langle \rho, c\rangle$ be an optimization problem. For any $x \in \{0,1\}^*$, let $opt(x)$ be the cost of optimal feasible solutions to x.

Definition 8 Algorithm M is an *ε-approximation* algorithm for $\langle \rho, c\rangle$, if for all (*sufficiently large*) $x \in \{0,1\}^*$:

- $x\ \rho\ M(x)$ — $M(x)$ is a feasible solution to x.
- $|c(M(x)) - opt(x)| / \max\{c(M(x)), opt(x)\} \leq \varepsilon$ — the relative difference between $M(x)$ and the optimal answer is bounded by some ε.

♢

Particular cases:

- A 0-approximation algorithm finds an optimal feasible solution. There is no difference from the optimal answer.
- A 1-approximation algorithm finds any feasible solution. The difference can be as big as possible, and it cannot be greater than 1 because $|c(M(x)) - opt(x)|$ cannot be greater than $\max\{c(M(x)), opt(x)\}$.

An optimization problem *can be ε-approximated* if it has a poly-time ε-approximation algorithm.

Example 1 *Vertex cover problem can be 1/2-approximized.*

Suppose that we are given a graph G where we need to solve vertex cover problem. The optimal solution would be the smallest vertex cover, so that this problem is a minimizing excercise. We propose an algorithm that finds an 1/2-approximated solution. It means that the answer of this algorithm does not exceed the optimal answer more than 2 times.

Algorithm: we will collect the vertex cover into a set K. Initially, K is empty. On each step, we take two arbitrary adjacent vertices u and v from the graph. Then we claim these vetices now belongs to the set K and remove them from the graph along with all the edges that connect to them. We repeat the step and take some other two vertices. We do it

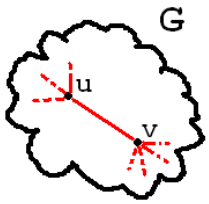

Figure 14: Taking two arbitrary vertices and deleting the edges

until all the edges have been removed from the graph. The set K is indeed a vertex cover, but it is not the optimal one. The size of K is however at most 2 times larger than the size of the optimal vertex cover, since at least one of the vertices u and v was really necessary for the vertex cover (otherwise, the edge between them would not be covered).

Example 2 *If travelling salesman problem can be ε-approximized with $\varepsilon < 1$, then* $\mathsf{P} = \mathsf{NP}$.

Suppose that we have a set of towns (probably a country). The towns are connected with roads. A travelling salesman lives in a certain town. He wants to visit all the other towns an return back to his hometown afterwards. The travelling salasman problem asks what is the shortest path that he has to travel by visiting each other town only once. This problem can be represented by an undirected weighted complete graph $G = \langle V, E \rangle$, where each vertex is a town. Let $n := |V|$. If there exists a road between cities u and v, we mark the corresponding edge with weight 1. If there is no road, we mark it with some $L > 1$. If the graph G has a Hamiltonian cycle, it means that there exist a salesman tour of of length n.

Now suppose that we can solve it in $K \cdot opt$ time for a fixed K. Construct the graph G as it is shown in Figure 15. If there is no road between two cities, we mark it with L, where $L := K \cdot n + 1$. By solving travelling salesman problem we can decide if the graph G has a Hamiltonian cycle.

- Hamiltonian $\Leftrightarrow$ $\exists$ way of length n, and the answer of our K-approximated algorithm is at most $K \cdot n$.
- Not Hamiltonian $\Leftrightarrow$ each way has length $\geq L + (n-1) = K \cdot n + 1 + n - 1 = K \cdot n + n = n \cdot (K + 1)$.

The answers are distinguishable! We can decide if the graph is Hamiltonian.

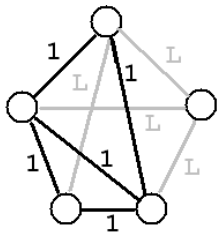

Figure 15: Travelling salesman problem as a graph

If a problem can be ε-approximized, then, according to the definition, it has a polynomial time approximization algoritm. We have solved the problem in polynomial time, and since deciding if the graph is Hamiltonian is NP-complete, we get that P = NP.

Various complexity classes

We can define special complexity classes for optimization problems.

- NPO — all optimization problems with poly-time ρ and c.
- APX — ε-approximable problems, where $\varepsilon < 1$. APX $\subsetneq$ NPO if we assume that travelling salesman is non-approximable (that P $\neq$ NP).
- PTAS — ε-approximable problems, for any $\varepsilon > 0$. Can be seperated from APX by defining artificial problems. For examle, given a propositional formula φ, find the evaluation such that φ is *true*. Feasible solution is *any* valuation of variables. Additionally, define the const function c: $c(x_1, \ldots, x_n) = 1$ iff $\varphi(x_1, \ldots, x_n) = \mathsf{true}$, otherwise $c(x_1, \ldots, x_n) = 2$. This problem belongs to APX, but does not belong to PTAS (can be approximized for some ε, but not for any ε).
  - The mapping $\varepsilon \to M_\varepsilon$ must be poly-time, where M_ε is the ε-approximation algorithm.
- FPTAS — there exists an algorithm $M(x, \varepsilon)$, such that
  - $M(\cdot, \varepsilon)$ is a ε-approximation algorithm.
  - Running time of $M(x, \varepsilon)$ is bounded by $q(|x|, 1/\varepsilon)$ for some polynomial q.

  It can be separated from PTAS by BIN PACKING problem, where objects of different volumes must be packed into a finite number of bins of a certain capacity in a way that minimizes the number of bins used. This problem can be approximized with any $\varepsilon > 0$, but the running time of the approximation algorithm is not polynomially bounded.

In conclusion, we get that FPTAS $\subseteq$ PTAS $\subseteq$ APX $\subseteq$ NPO. All inclusions are strict unless P = NP.