Computationally Hard Problems

Dynamic Programming and Approximation Algorithms

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

Fall 2020

The Approximation Scheme

Given: objects $a_1, \ldots, a_n$ with weights $w_1, \ldots, w_n$ and values $s_1, \ldots, s_n$ and capacity B.

Step 1: Let

$$k := \frac{\epsilon \cdot \max\{s_1, \ldots, s_n\}}{(1+\epsilon)n}$$

and create a modified instance to the knapsack problem with the original weights $w_1, \ldots, w_n$ but modified values $\tilde{s}_1, \ldots, \tilde{s}_n$, where

$$\tilde{s}_i := \left\lfloor \frac{s_i}{k} \right\rfloor = \left\lfloor \frac{(1+\epsilon) \cdot n \cdot s_i}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor$$

for $1 \leq i \leq n$.

Moreover, capacity B is the same as in the original instance.

Step 2: Run second pseudo-polynomial algorithm (the one with the $n \times S$ table) on modified instance and use its solution for original problem. Why can we do that?

Analyzing the Running Time

Modified instance can be created in linear time. Running time of pseudo-polynomial algorithm is

$$O(n \cdot (\tilde{s}_1 + \cdots + \tilde{s}_n)) = O(n \cdot n \cdot \max\{\tilde{s}_1, \ldots, \tilde{s}_n\}),$$

which, by definition of the $\tilde{s}_i$, is

$$O\left(n^2 \cdot \max\left\{\left\lfloor \frac{(1+\epsilon)\cdot n \cdot s_1}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor, \ldots, \left\lfloor \frac{(1+\epsilon)\cdot n \cdot s_n}{\epsilon \cdot \max\{s_1, \ldots, s_n\}} \right\rfloor \right\}\right).$$

This the same as

$$O\left(n^2 \cdot \frac{(1+\epsilon)\cdot n}{\epsilon}\right) = O\left(\frac{n^3}{\epsilon} + n^3\right).$$

Altogether, running time is polynomial in the input length n and in $1/\epsilon$. If ϵ is a constant, then the running time is $O(n^3)$.

Analyzing the Approximation Quality (1/3)

Notation:

- A_{mod}: optimal solution to modified instance (computed)
- A_{org}: optimal solution to original instance (not computed!)

Both solutions are legal (do not exceed capacity). Since A_{mod} optimal for modified instance,

$$\sum_{a_i \in A_{\text{org}}} \tilde{s}_i \leq \sum_{a_i \in A_{\text{mod}}} \tilde{s}_i \qquad (*)$$

Algorithm outputs: $\sum_{a_i \in A_{\text{mod}}} s_i$. (Would like to know $\sum_{a_i \in A_{\text{org}}} s_i$, but cannot compute in the given time.)

Goal: compare to $\mathrm{OPT}_{\text{org}} = \sum_{a_i \in A_{\text{org}}} s_i$. Use also:

$$\frac{s_i}{k} - 1 \leq \tilde{s}_i \leq \frac{s_i}{k} \qquad (**)$$

Analyzing the Approximation Quality (2/3)

Using $(*)$ and $(**)$,

$$\sum_{a_i \in A_{\text{mod}}} s_i \overset{(**)}{\geq} k \cdot \sum_{a_i \in A_{\text{mod}}} \tilde{s}_i \overset{(*)}{\geq} k \cdot \sum_{a_i \in A_{\text{org}}} \tilde{s}_i$$

$$\overset{(**)}{\geq} k \cdot \sum_{a_i \in A_{\text{org}}} \left(\frac{s_i}{k} - 1 \right) \geq \left(\sum_{a_i \in A_{\text{org}}} s_i \right) - nk = \text{OPT} - nk$$

Expanding k, the last expression is

$$\text{OPT} - n \cdot \frac{\epsilon \cdot \max\{s_1, \ldots, s_n\}}{(1+\epsilon)n} = \text{OPT} - \frac{\epsilon}{1+\epsilon} \cdot \max\{s_1, \ldots, s_n\}.$$

Clearly, $\text{OPT} \geq \max\{s_1, \ldots, s_n\}$ (note that $w_i \leq B$ for all i).

Analyzing the Approximation Quality (3/3)

Altogether,

$$\sum_{a_i \in A_{\text{mod}}} s_i \geq \text{OPT} - \frac{\epsilon}{1+\epsilon} \cdot \text{OPT} = \text{OPT} \cdot \left(1 - \frac{\epsilon}{1+\epsilon}\right) = \frac{\text{OPT}}{1+\epsilon}$$

We have proved

$$\frac{\text{OPT}}{\sum_{a_i \in A_{\text{mod}}} s_i} \leq 1 + \epsilon,$$

hence our solution is really by at most a factor $1 + \epsilon$ off. $\square$

Computationally Hard Problems

Design of Randomized Algorithms: Independent Set

Carsten Witt

Institut for Matematik og Computer Science
Danmarks Tekniske Universitet

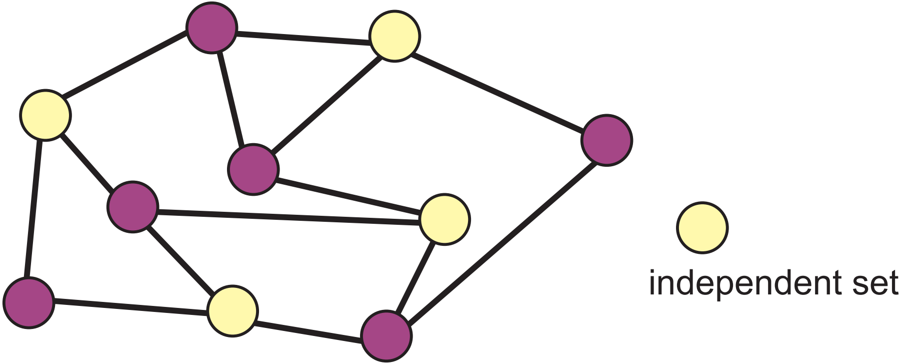
Fall 2020

The Independent Set Problem

Definition For an undirected graph $G = (V, E)$ an *independent set* is a subset $V' \subseteq V$ of the vertices such that no edges are present, i. e., for all $v, w \in V'$, $v \neq w$: $\{v, w\} \notin E$.

Problem $[\textsc{IndependentSet}]$ **Input:** An undirected graph $G = (V, E)$. In the decision version, also a natural number k.
Output (decision version): YES if the graph G has an independent set of size k and NO otherwise.
Output (optimization version): An independent set of maximal size.

Example



The Independent Set Problem

The Independent Set Problem is a “hard” problem.

It is $\mathcal{NP}$-complete, i. e., any algorithm solving the problem probably needs superpolynomial time.

One cannot do much better to find a maximum independent set than to test all subsets of V.

Instead we look again at an approximate solution, i. e., a *large* – but not necessarily *largest* – independent set.

We design a randomized algorithm and derive a probabilistic guarantee for the approximation quality.

The Algorithm

Let $G = (V, E)$, and $V = \{1, 2, \ldots, n\}$, and $m = |E|$

We assign a real number p_i, $0 \leq p_i \leq 1$ to every node i. How, is decided by the user; examples follow later.

Note: The p_i need not form a probability distribution, i. e., $\sum_{i=1}^{n} p_i \neq 1$ is possible.

The algorithm has two stages:

1. Some nodes are selected into a set I.
2. Nodes (and the incident edges) are removed from I to make it an independent set.

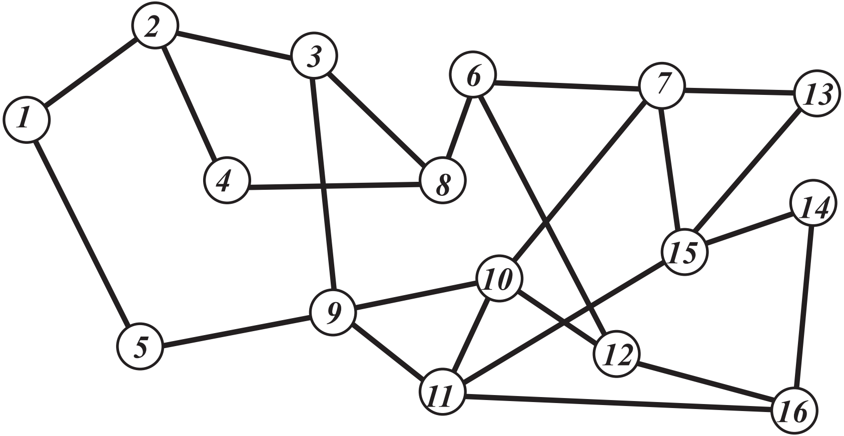
Stage 1

- Initialize a set $I = \emptyset$.
- For every vertex i independently with probability p_i put i into I (i. e., with probability $1 - p_i$ vertex i is not put into I).
- Every node i contributes 1 to the size of I with probability p_i. The expected size of I is

$$\boldsymbol{E}\left[\,|I|\,\right] = \sum_{i=1}^{n} 1 \cdot p_i = \sum_{i=1}^{n} p_i$$

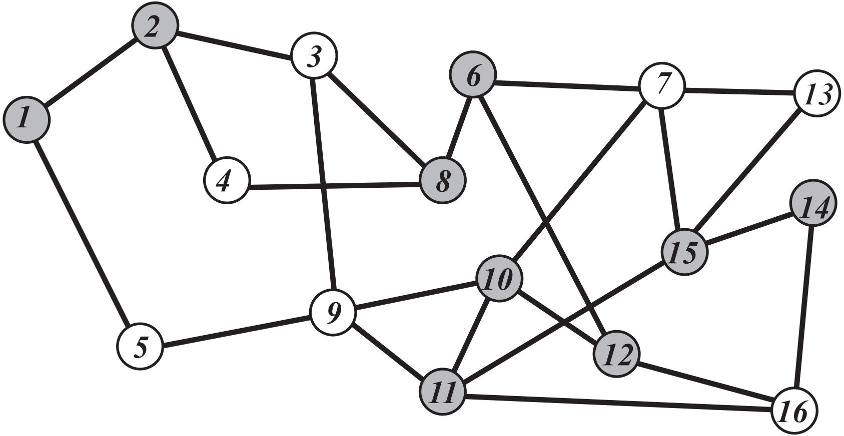
(for details: size of a random selection, A.2 in lecture notes)

Example Stage 1



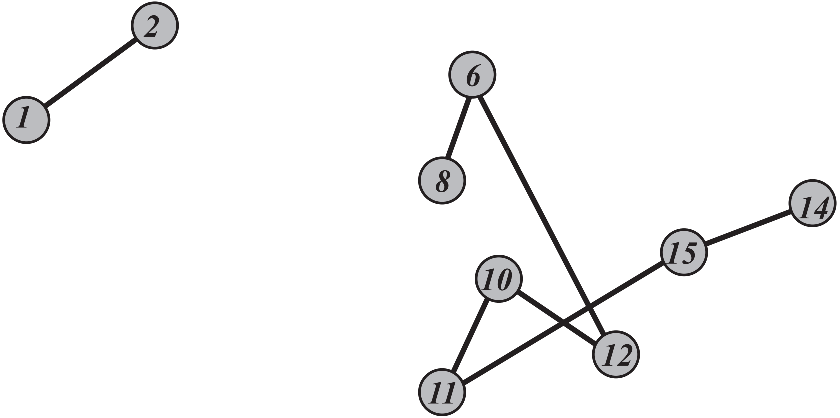
The original graph.

Example Stage 1



The gray nodes are selected into the set I.

Example Stage 1



We can "forget" the non-selected nodes $V \setminus I$ and incident edges.

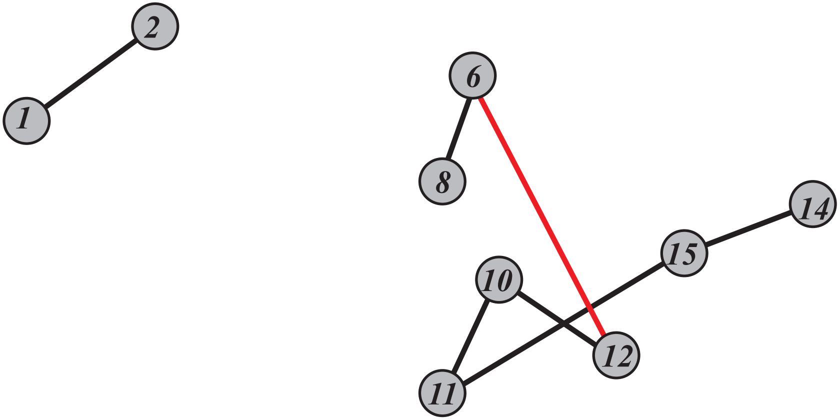
Stage 2

For every edge $\{i, j\} \in E$ we check whether both end-nodes are in I.

If this is the case, remove one of i or j from I.

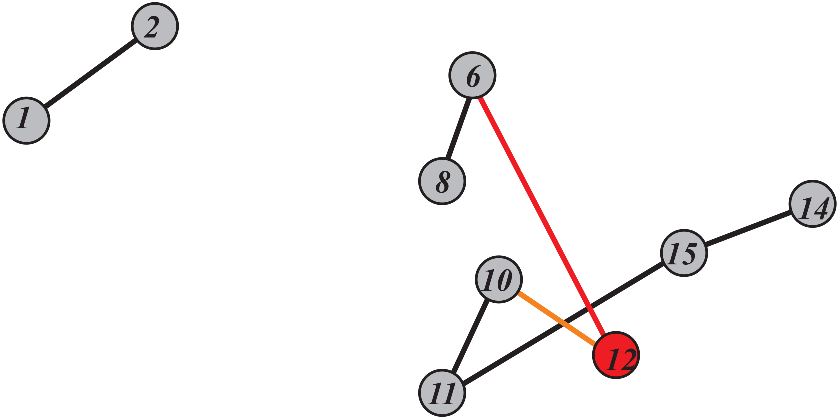
Also remove all edges which are incident on a removed node.

Example Stage 2



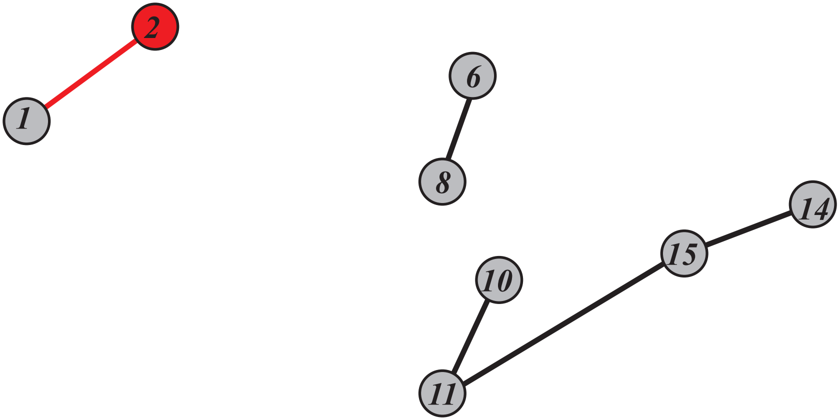
Edge $\{6, 12\}$ has both end-nodes in I.

Example Stage 2



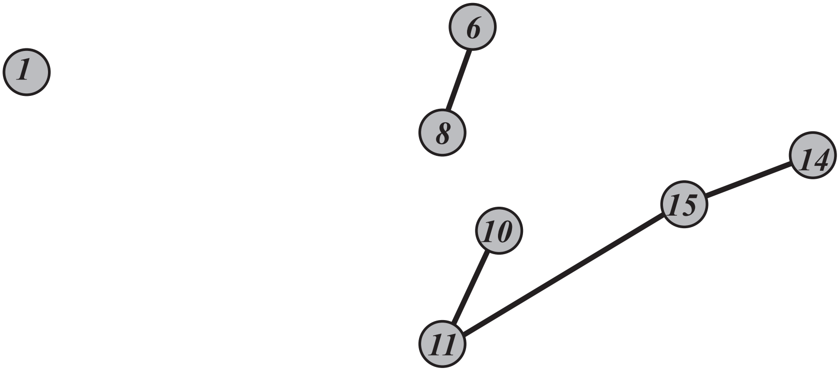
We (randomly) choose to remove node 12 and all incident edges.

Example Stage 2

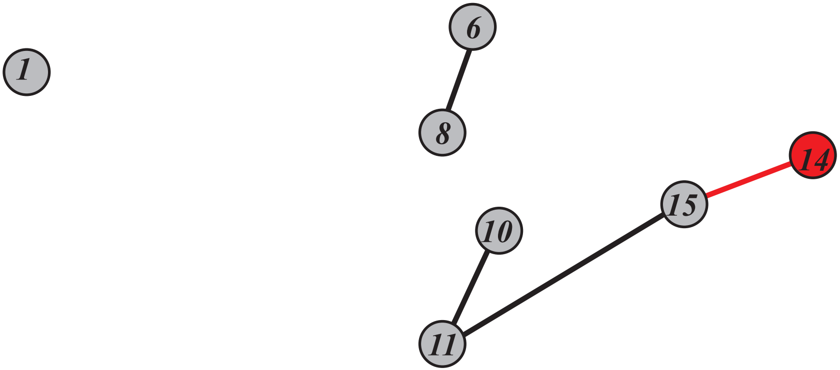


And so on ...

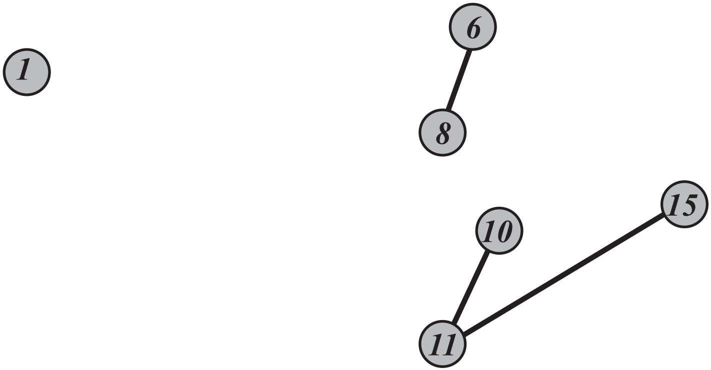
Example Stage 2



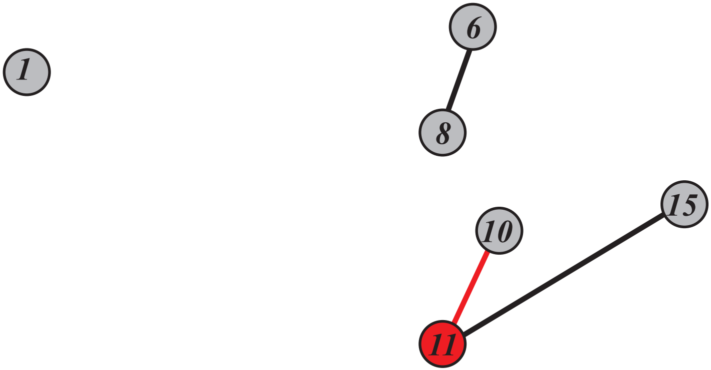
Example Stage 2



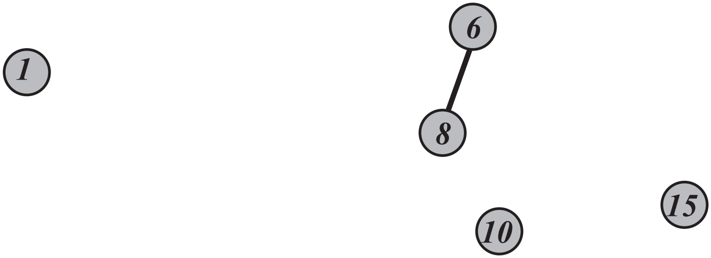
Example Stage 2



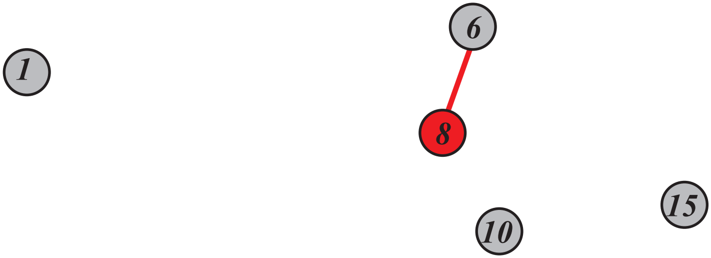
Example Stage 2



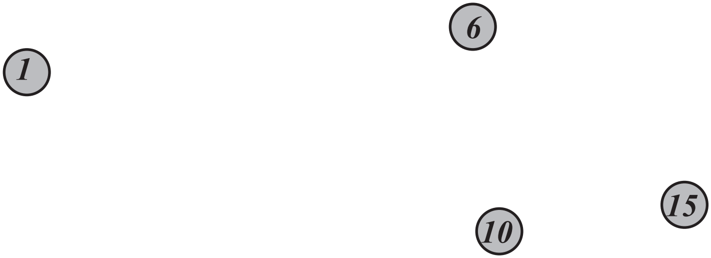
Example Stage 2



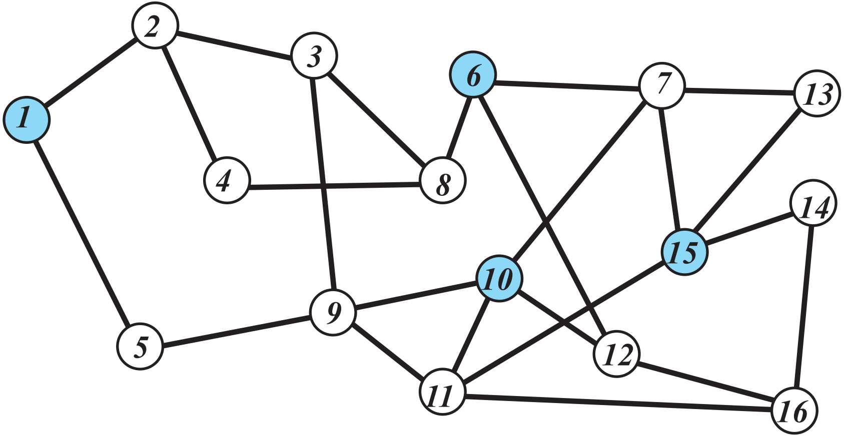
Example Stage 2



Example Stage 2

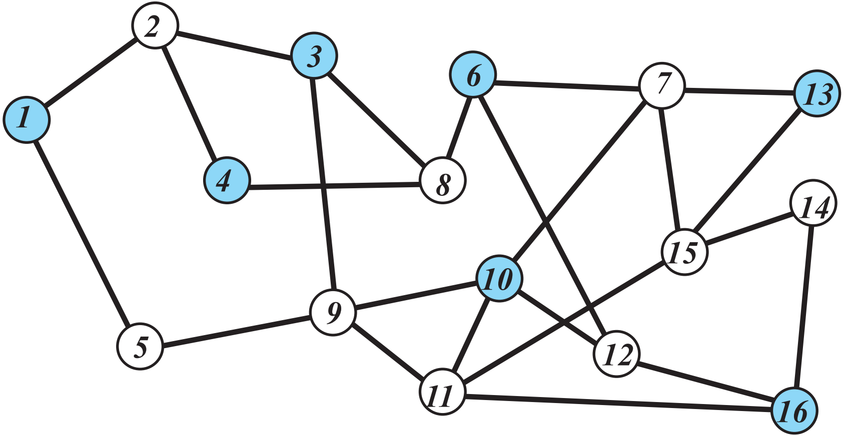


Example Stage 2



The independent set found is shown in blue.

Example Stage 2



The set in the example is not maximum; here is a larger one.

Running Time Analysis

$I \leftarrow \emptyset$
for $i = 1, \ldots, n$ **do**
  with prob p_i: $I \leftarrow I \cup \{i\}$
end for
for $\{i, j\} \in E$ **do**
  if $((i \in I) \wedge (j \in I))$ **then**
    if $(\textbf{coin} = 0)$ **then**
      $I \leftarrow I \setminus \{i\}$
    else
      $I \leftarrow I \setminus \{j\}$
    end if
  end if
end for
return I

The running time is $O(|V| + |E|)$.

Performance of the Algorithm

In the first stage, the set I is constructed. We showed that the expected size of I is

$$\boldsymbol{E}\left[\,|I|\,\right] = \sum_{i=1}^{n} p_i$$

Now we analyze how much of this is left (on average) after deleting nodes from I.

Performance of the Algorithm

- In the second stage nodes are deleted from I.
- Let F be the set of edges "in" I at the beginning of Stage 2, where an edge $\{i, j\}$ is said to be in F if both endpoints i, j are in I.
- At most one node is deleted for every edge in F.
- Let $\{i, j\}$ be an edge in E.
- i gets into I in Stage 1 with prob. p_i.
- j gets into I in Stage 1 with prob. p_j.
- Both get into I with prob. $p_i p_j$.
- Edge $\{i, j\}$ is in F with prob. $p_i p_j$.

Performance of the Algorithm

$$\boldsymbol{E}\left[\text{number of nodes removed from } I\right]$$

$$\leq \boldsymbol{E}\left[|F|\right] = \sum_{\{i,j\}\in E} p_i p_j$$

Performance of the Algorithm

Let I^* denote the (random) independent set output by the algorithm.

$$\begin{aligned} \boldsymbol{E}\left[\,|I^*|\,\right] &= \boldsymbol{E}\left[\,\#(\text{nodes put into } I)\,\right] - \boldsymbol{E}\left[\,\#(\text{nodes removed from } I)\,\right] \\ &\geq \sum_{i=1}^{n} p_i - \sum_{\{i,j\}\in E} p_i p_j \end{aligned}$$

How should one choose the p_i to maximize this expression?

One possible approach is to $p_1 = \cdots = p_n = p$.

Which value should we choose for p?

$$\boldsymbol{E}\left[\,|I^*|\,\right] \geq \sum_{i=1}^{n} p - \sum_{\{i,j\}\in E} p\,p = np - mp^2 \; .$$

Performance of the Algorithm

$$\boldsymbol{E}\left[\,|I^*|\,\right] \geq np - mp^2 \ .$$

For fixed n and m this a function of p.

Let us maximize it! (by differentiating)

$$\begin{aligned} f(p) &= np - mp^2 \\ f'(p) &= n - 2mp \end{aligned}$$

Solving $n - 2mp = 0$ for p gives $p = n/(2m)$. This is a maximum.

Note $n/(2m) = 1/d_{\mathsf{avg}}$, where d_{avg} is the average degree of G.

Summary

Plugging $p = n/(2m)$ into the formula $\boldsymbol{E}[\,|I^*|\,] \geq np - mp^2$ gives

$$\boldsymbol{E}[\,|I^*|\,] \;\geq\; \frac{n^2}{2m} - m\frac{n^2}{4m^2} \;=\; \frac{n^2}{2m} - \frac{n^2}{4m} \;=\; \frac{n^2}{4m} \;=\; \frac{n}{2d_{\mathsf{avg}}}.$$

The randomized algorithm above finds an independent set of *expected* size at least $n/(2d_{\mathsf{avg}})$.

Other choices for the p_i are possible (when the structure of G is known), for example

$$p_i = \frac{1}{d_i + 1} \text{ where } d_i \text{ is the degree of } i$$

Expected Approximation Ratio

Let $A(G)$ be the size of the independent set produced by our algorithm on G.

Let $\mathrm{OPT}(G)$ be the maximal size of an independent set in G.

As defined before, the approximation ratio is

$$R_A(G) = \frac{\mathrm{OPT}(G)}{A(G)} \; .$$

If the algorithm is randomized then $R_A(G)$ is defined to be the *expected approximation ratio*:

$$R_A(G) = \frac{\mathrm{OPT}(G)}{\boldsymbol{E}\,[\,A(G)\,]} \; .$$

Expected approximation ratio is no worse than $\frac{n}{n/(2d_{\mathsf{avg}})} = 2d_{\mathsf{avg}}$.

Using the Algorithm

Let $G = (V, E)$ be the graph in which we want to find an independent set.

- Determine the probabilities p_i.
- Compute the bound on the expected size $B = \sum_{i=1}^{n} p_i - \sum_{\{i,j\} \in E} p_i p_j$
- Run the algorithm; let I^* be the independent set found.
- If $|I^*| \geq B$ accept the set and stop;
  otherwise run the algorithm again.
- If the algorithm does not find an independent set of size at least B in the time bound you have, then take the largest one you have found. You have been unlucky.

The Problem

Problem [MaximumSatisfiability]

Input: A set of clauses over n boolean variables (and a natural number k for the decision version).

Output for the decision version: YES if there is an assignment of truth values to the variables which satisfies at least k clauses and NO otherwise.

Output for the optimizing version: An assignment of truth values to the variables which satisfies a maximal number of clauses.

A Simple Randomized Solution

Let $C = \{c_1, c_2, \ldots, c_m\}$ be a set of clauses over the Boolean variables $x_1, x_2, \ldots, x_n$.

We want to show that there is an assignment of truth values to the x_i which satisfies at least $m/2$ clauses.

We assign truth values to the variables at random.

```
for i = 1, ..., n do
    if (coin = 0) then
        x_i ← false
    else
        x_i ← true
    end if
end for
```

A Simple Randomized Solution

Let $c_j = x_1 \vee \cdots \vee x_{k_j}$ a clause.

The prob. that c_j is **not** satisfied by the random assignment is $(1/2)^{k_j}$. Why?

For every literal of c_j the probability to be not satisfied is $(1/2)$.

The probability that all literals of c_j are not satisfied is $(1/2)^{k_j}$.

The probability that some literal of c_j is satisfied is $1 - (1/2)^{k_j}$.

The probability s_j that c_j is satisfied is

$$s_j = 1 - (1/2)^{k_j}$$

Observation: Longer clauses are more easily satisfied.
Observation: For every clause c_j

$$s_j = 1 - (1/2)^{k_j} \geq 1 - (1/2) \geq 1/2.$$

Let N be the number of clauses satisfied.

N is a random variable.

The expected value is

$$\boldsymbol{E}\,[\,N\,] = \sum_{j=1}^{m} s_j .$$

Using the observation this gives

$$\boldsymbol{E}\,[\,N\,] \geq \sum_{j=1}^{m} \frac{1}{2} = \frac{m}{2}.$$

Since $\boldsymbol{E}\,[\,N\,] \geq m/2$, there must be at least one assignment that satisfies at least $m/2$ clauses. Otherwise the expectation would be smaller (Fact A.1 in lecture notes, *probabilistic method*).

Approximation Ratio

Let $A(C)$ be the number of clauses from $C = \{c_1, c_2, \ldots, c_m\}$ satisfied by algorithm A.

Let $\mathrm{OPT}(C)$ be the maximal number of clauses that can be satisfied.

Recall that if A is randomized then $R_A(C)$ is defined to be the *expected approximation ratio*:

$$R_A(C) = \frac{\mathrm{OPT(C)}}{\boldsymbol{E}\left[\, A(C) \,\right]} \; .$$

The last algorithm has an expected approximation ratio of 2.

Outline

The aim is to improve the ratio (to $4/3$).

The above algorithm has a bad performance on short clauses.

It is good on long clauses.

We describe a different algorithm based on formulating $\textsc{MaxSat}$ as linear program.

We run both algorithms.

Finally we select the better solution.

Integer Programming

Problem [IntegerProgramming]
Input: Parameters $c_1, \ldots, c_m \in \mathbb{Z}$, $a_{j1}, \ldots, a_{jm}, b_j \in \mathbb{Z}$, $j = 1, \ldots, k$.
Output for the optimizing version: Values $x_1, \ldots, x_m \in \mathbb{Z}$ such that

- The target function $c_1 x_1 + \cdots + c_m x_m$ is maximized.
- The constraints are met $a_{j1} x_1 + \cdots + a_{jm} x_m \leq b_j, \quad j = 1, \ldots, k$

The problem is $\mathcal{NP}$-hard. If the numbers x_i are allowed to be reals then the problem is efficiently solvable.