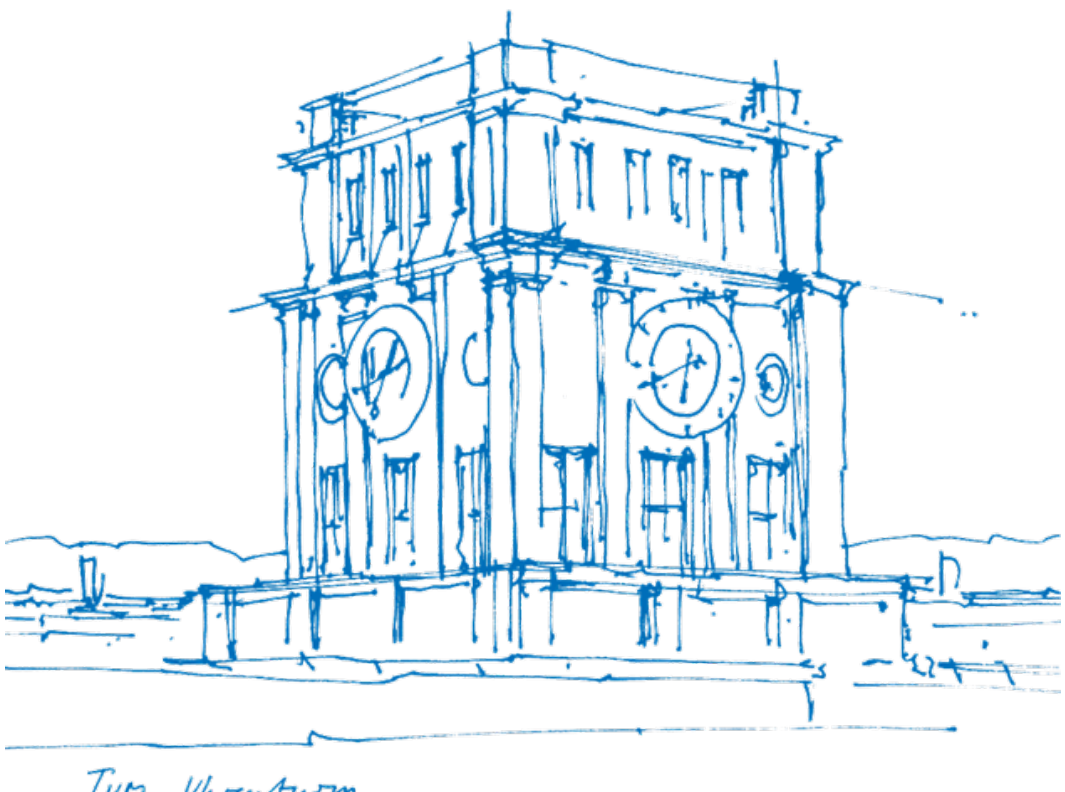
School of Computation, Information and Technology
Technical University of Munich

AACPP 2025

Week 11: NP-hardness



Mateusz Gienieczko, **Mykola Morozov**

School of Computation, Information and Technology
Technical University of Munich

2025.07.15

Eight round – survey

Ninth – final! – round

Deadline – 22.07.2025, 10:00 AM.

Last task!

PPM – Puzzle of the Perfect Meows

Given a weight function on the alphabet

$$w : \Sigma \to \mathbb{N}^+, \Sigma = \{1, ..., k\}$$

construct the cheapest prefix-free code C of n words (cheapest sum of weights of all letters of all words).

Prefix-free means that:

$$\forall_{w \in C}. \neg \exists_{u \in C}. u \sqsubset w$$

Prefix-free codes

Prefix-free codes can be unambiguously decoded.

A trie of a prefix-free code has terminal nodes only in leaves.

So we can decode by walking down the tree, accepting in leaves, going back to root.

If a code is not prefix-free then there is ambiguity (in an accepting node do we accept and go to root, or do we continue going down?).

PPM – Puzzle of the Perfect Meows

We can translate the task into constructing the cheapest trie with n leaves.

PPM – Puzzle of the Perfect Meows

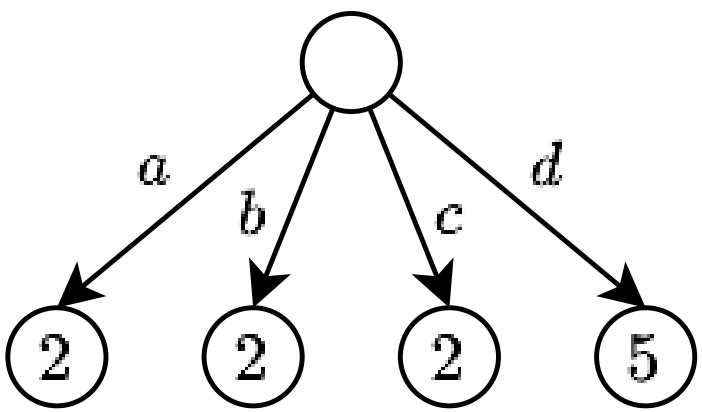
We can translate the task into constructing the cheapest trie with n leaves.

The obvious greedy strategy that comes to mind is – for a tree of i nodes add the leaf that costs the least and obtain the tree on $i + 1$ nodes.

For a node that is a leaf we need to create two children with the cheapest letters.

This doesn't work.

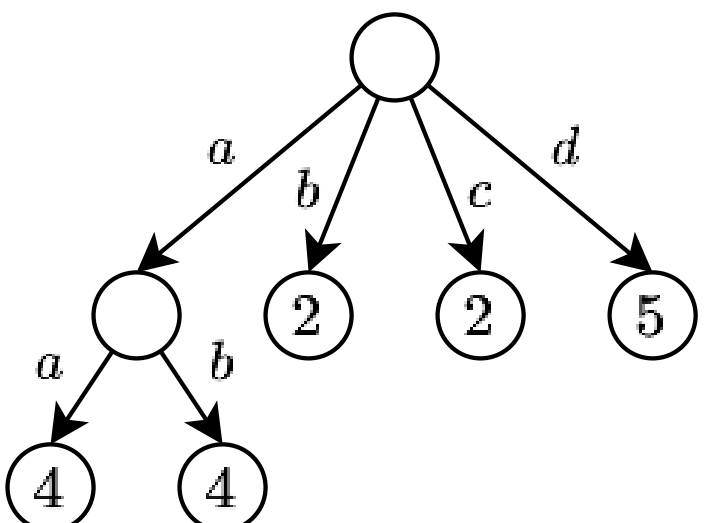
PPM – Puzzle of the Perfect Meows



$\Sigma = \{a, b, c, d\}, w(a) = 2, w(b) = 2, w(c) = 2, w(d) = 4$

For $n = 4$ the optimal tree has cost 11.

PPM – Puzzle of the Perfect Meows

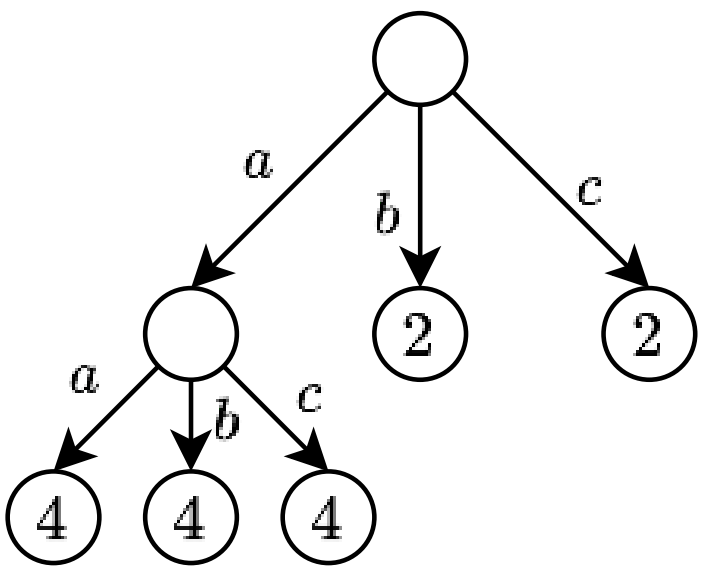


$\Sigma = \{a, b, c, d\}, w(a) = 2, w(b) = 2, w(c) = 2, w(d) = 4$

For $n = 4$ the optimal tree has cost 11.

For $n = 5$, the greedy algorithm splits one of the leaves at depth 2 creating a tree of cost 17...

PPM – Puzzle of the Perfect Meows



$\Sigma = \{a, b, c, d\}, w(a) = 2, w(b) = 2, w(c) = 2, w(d) = 4$

For $n = 4$ the optimal tree has cost 11.

For $n = 5$, the greedy algorithm splits one of the leaves at depth 2 creating a tree of cost 17...

But the optimal tree here has cost 16!

This problem does not have *optimal substructure*.

PPM – Puzzle of the Perfect Meows

Correct solution – take the cheapest current leaf and add *all* of its possible children.

Phase 1: continue until the number of leaves is $\geq n$.

Phase 2: then take the cheapest leaf and see if adding some of its children wouldn't improve the cost. Continue while cost improves.

PPM – Puzzle of the Perfect Meows

Why does this work?

Lemma: *In an optimal trie of n leaves there cannot be an internal (non-leaf) node v of depth (sum of weights on path to root) higher than that of some leaf u.*

PPM – Puzzle of the Perfect Meows

Why does this work?

Lemma: *In an optimal trie of n leaves there cannot be an internal (non-leaf) node v of depth (sum of weights on path to root) higher than that of some leaf u.*

Proof: If it were the case, then we could take the subtree S (non-empty by assumption) of the internal node and "transplant" it as the subtree of the leaf. This makes v a leaf and changes the overall cost of the tree by ($r(S)$ is the number of leaves in S):

$$d(v) - d(u) + r(S)d(u) - r(S)d(v)$$

which is negative if $d(u) < d(v)$ and $r(S) > 1$.

PPM – Puzzle of the Perfect Meows

The algorithm has two phases – expansion to n leaves and then cost improvement.

In the first phase, since we always develop the cheapest leaf, we don't violate the condition from the Lemma.

Cost improvement cannot make anything worse – it always leaves us with a tree with n leaves that does not violate the Lemma.

We need to show that if there are no more possible improvements then the tree is optimal.

PPM – Puzzle of the Perfect Meows

Assume U is optimal and T is a tree in phase two of our algorithm. We assume $w(T) > w(U)$ and we will show that the cost reduction rule applies.

Take the cheapest word u that is in U but is not in T. Such a word must exist, since $w(U) < w(T)$. By the Lemma, in U only leaves have weight $> w(u)$.

If u is not a prefix of any word in T then we can take the deepest leaf of T and apply the cost reduction rule.

PPM – Puzzle of the Perfect Meows

We now show u cannot be a prefix of a word in T.

PPM – Puzzle of the Perfect Meows

We now show u cannot be a prefix of a word in T. If a, b are the cheapest letters of Σ, then the node u in T has to have at least the children ua and ub. Observe that in U the depth of all leaves has to be at least:

$$w(u) + w(a) + w(b)$$

or else we could apply the cost reduction rule to U, which is the optimal tree.

PPM – Puzzle of the Perfect Meows

We now show u cannot be a prefix of a word in T. If a, b are the cheapest letters of Σ, then the node u in T has to have at least the children ua and ub. Observe that in U the depth of all leaves has to be at least:

$$w(u) + w(a) + w(b)$$

or else we could apply the cost reduction rule to U, which is the optimal tree.

Moreover, all of those words have to also be in T, because they could only be removed *after* ua and ub in the second phase (since they have higher weight).

But then we have $|U \cap T| = n - 1$ and $\{ua, ub\} \in T \setminus U$, which gives $|T| = n + 1$. In other words, if u were expanded in T then we would have too many leaves.

PPM – Puzzle of the Perfect Meows

It remains to show this algorithm terminates relatively quickly.

The key observation is that if a node is expanded in the second phase then its two lightest children are never removed (it does not become a leaf again).

PPM – Puzzle of the Perfect Meows

It remains to show this algorithm terminates relatively quickly.

The key observation is that if a node is expanded in the second phase then its two lightest children are never removed (it does not become a leaf again).

This bounds the number of iterations by nk, since every node can be expanded at most once and k children added to it.

PPM – Puzzle of the Perfect Meows

If a node v was expanded then it had some at-the-time minimal depth d, while the maximal leaf with depth h was removed, and this was cost-negative:

$$d + w(a) + w(b) < h$$

For the heavier child to be removed, $d + w(b)$ would have to become the new maximum, and a node u of depth d' expanded so that:

$$d' + w(a) + w(b) < d + w(b)$$

In particular, $d' < d$. But this cannot happen, because then our algorithm would've expanded u *before* v.

PPM – Puzzle of the Perfect Meows

```
fn improve()
  s_ls = leaves()
  s_c = cost_sum()
  min_leaf = pop_min()
  add_leaf(min_leaf.0 + w[0])
  i = 1
  if s_ls < n { // Phase 1
    while i < k && leaves() < n
      new_l = min_leaf + w[i];
      add(new_l)
      i += 1
  }
  while i < k { // Phase 2
    new_l = min_leaf + w[i]
    if new_l < peek_max()
      pop_max()
      add(new_l)
      i += 1
    else
      break
  }

  return s_ls < n ||
         s_c > cost_sum()
```

PPM – Puzzle of the Perfect Meows

We need two heaps (min and max) and to maintain the number of leaves and the current cost.

Since you can't remove an arbitrary key from the heap, we need to keep track of which weights are active in an auxiliary table or map.

Total cost is $\mathcal{O}(n + k)$ memory and $\mathcal{O}(nk \log n)$ time.

Recall the plan

- Greedy and dynamic programming (DP)
- Trees
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- Ways to run DP on graphs (Toposort)
- Advanced graph algorithms (Matchings, flows)
- Binary Search Trees
- Number theory
- String algorithms (KMP, tries, suffix tables)
- **Some problems can't* even be solved efficiently (NP-completeness)**

  ↑ *we are here*

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
-
-
-
-
-
-

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
-
-
-
-
-

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
- What is the smallest vertex cover? (set of vertices that touches all edges)
-
-
-
-

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
- What is the smallest vertex cover? (set of vertices that touches all edges)
- Can this graph be 3-coloured? (each vertex assigned one of three colours such that no two vertices sharing an edge have the same colour)
-
-
-

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
- What is the smallest vertex cover? (set of vertices that touches all edges)
- Can this graph be 3-coloured? (each vertex assigned one of three colours such that no two vertices sharing an edge have the same colour)
- Does this graph contain a path on n vertices? (Hamiltonian path)
-
-

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
- What is the smallest vertex cover? (set of vertices that touches all edges)
- Can this graph be 3-coloured? (each vertex assigned one of three colours such that no two vertices sharing an edge have the same colour)
- Does this graph contain a path on n vertices? (Hamiltonian path)
- Does this graph contain a cycle on n vertices? (Hamiltonian cycle)
-

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
- What is the smallest vertex cover? (set of vertices that touches all edges)
- Can this graph be 3-coloured? (each vertex assigned one of three colours such that no two vertices sharing an edge have the same colour)
- Does this graph contain a path on n vertices? (Hamiltonian path)
- Does this graph contain a cycle on n vertices? (Hamiltonian cycle)
- What is the longest path in this graph?

Hard graph problems

- What is the biggest clique? (set of vertices where each pair is connected)
- What is the biggest independent set? (set of vertices where *none* are connected)
- What is the smallest vertex cover? (set of vertices that touches all edges)
- Can this graph be 3-coloured? (each vertex assigned one of three colours such that no two vertices sharing an edge have the same colour)
- Does this graph contain a path on n vertices? (Hamiltonian path)
- Does this graph contain a cycle on n vertices? (Hamiltonian cycle)
- What is the longest path in this graph?

Turns out we don't know how to solve any of those in polynomial time.

P vs NP

Let us first give an informal but intuitive characterisation of P and NP.

We are talking about *decision problems*, where the answer is **YES** or **NO**.

P vs NP

Let us first give an informal but intuitive characterisation of P and NP.

We are talking about *decision problems*, where the answer is **YES** or **NO**.

The class P contains problems that can be solved in polynomial time wrt. to the input size.

P vs NP

Let us first give an informal but intuitive characterisation of P and NP.

We are talking about *decision problems*, where the answer is **YES** or **NO**.

The class P contains problems that can be solved in polynomial time wrt. to the input size.

The class NP contains problems to which *a polynomial witness* can be *verified* in polynomial time (wrt. the sum of input and output sizes).

For example, if the problem is “is there a three-colouring of this graph”, and I give you the colouring, it can be validated in linear time (check each edge).

NP problems can be solved in exptime

From the definition, any problem in NP can be solved in time exponential in the input length.

Just generate all witnesses and verify them.

Unfortunately, NP-hard problems appear all over the place, so we need to cope with them.

NP-hard

A problem is *NP-hard* if it is “at least as hard as the hardest problems in NP”.

This is determined by *reductions*. Solving a problem X reduces to the problem Y if an algorithm solving Y can be easily[1] converted to an algorithm solving X.

For example, solving Max Independent Set in a graph G can be reduced to solving Max Clique in $G^{\complement}$.

A problem is *NP-complete* if it is NP-hard and also in NP. Intuitively, knowing a fast algorithm for an NP-c problem immediately gives algorithms for *all* NP problems “for free” .

[1] In P or L. L is slightly weaker, but we don’t actually know if it makes a difference.

More formally

NP is the problems solvable on a nondeterministic Turing Machine in polynomial time, while P requires determinism.

A nondeterministic TM can “guess” a witness and verify it. A deterministic TM cannot “guess”. This is why nondeterministic TMs *seem* more powerful.

The platonic NP-hard problem is “Given a nondeterministic TM and an instance of a problem, determine if it accepts in polynomial time”.

Since every NP-complete problem could be used as a reduction for the above, if NP = P then there would be no difference between deterministic and nondeterministic algorithms (in poly time), which sounds weird.

Examples of reductions

We mentioned the Clique – Independent Set reduction.

Examples of reductions

We mentioned the Clique – Independent Set reduction.

There is an obvious duality with Min Cover and Max Independent Set.

Examples of reductions

We mentioned the Clique – Independent Set reduction.

There is an obvious duality with Min Cover and Max Independent Set.

Hamiltonian Cycle is easily reducible to Hamiltonian Path, which in turn reduces to Longest Path.

Examples of reductions

We mentioned the Clique – Independent Set reduction.

There is an obvious duality with Min Cover and Max Independent Set.

Hamiltonian Cycle is easily reducible to Hamiltonian Path, which in turn reduces to Longest Path.

For something less obvious, 3-colouring can be reduced to Independent Set.

“Non-graph” problems

Knapsack problem is NP-hard.

0-1 Integer Programming is NP-hard.

SAT and 3-SAT are NP-hard.

Coping with NP-Completeness

Sometimes, *approximations* are possible.

For example, Metric Travelling-Salesman can be approximated quite well...

Coping with NP-Completeness

Sometimes, *approximations* are possible.

For example, Metric Travelling-Salesman can be approximated quite well...

But general Travelling-Salesman cannot be approximated at all.

We can 2-approximate vertex cover...

Coping with NP-Completeness

Sometimes, *approximations* are possible.

For example, Metric Travelling-Salesman can be approximated quite well…

But general Travelling-Salesman cannot be approximated at all.

We can 2-approximate vertex cover…

But that's pretty useless if the cover is big. Also it's done with the dumbest algorithm you can think of (just find the max matching and return that).

Coping with NP-Completeness

Sometimes, *approximations* are possible.

For example, Metric Travelling-Salesman can be approximated quite well…

But general Travelling-Salesman cannot be approximated at all.

We can 2-approximate vertex cover…

But that's pretty useless if the cover is big. Also it's done with the dumbest algorithm you can think of (just find the max matching and return that).

Sometimes happen in competitive programming, but usually an exact solution is expected.

How I Learnt to Stop Worrying and Love the Backtrack

Backtracking is a quite direct way of implementing an NP algorithm.

We try to follow every possible sequence of choices. If it doesn't lead to a solution, we *backtrack* the choice and choose differently.

For example, we might choose any vertex into a vertex cover. So, choose one, remove all its edges, and try to solve on the smaller graph. After that, backtrack and make the opposite choice – choose all its neighbours – and solve again.

This is naturally exponential (for example on a path).

Backtracking

When backtracking we fight for each possible reduction in the search space.

Sometimes memoising the results like in usual recursive DPs gives speedups.

Optimising the time spent in each step is also crucial. For example, the classic algorithm of Knuth for exact cover benefits greatly from efficiently removing columns and rows in a matrix.

Parameterised complexity

Some NP-hard problems can be solved "with a parameter".

For example, Clique is NP-hard. But "is there a clique of size k" is solvable in $\mathcal{O}\left(n^k\right)$, which might be fine if k is suitably small (like 3).[2]

Clique is actually pretty bad to parameterise – it's like we *can't* do better than $\mathcal{O}\left(n^k\right)$ in general. But if we add a restriction on the max degree in the graph Δ, then we can find an $\mathcal{O}\left(2^\Delta \Delta^2 n\right)$ algorithm.

All of those are paramaterised algorithms, where the complexity depends on some parameter that depends on the input, but is not just its size.

[2] For $k = 3$ we can do better on sparse graphs ($\mathcal{O}(nm)$).

Kernelisation

Sometimes when the parameter we're considering is small, but the input size is big, we can shrink the input to an equivalent instance.

If the result of this pre-processing is small (bounded by some reasonable function of the parameter), then we call this *kernelisation* and the smaller instance a *kernel*.

Intuitively, it's the "hard part" of the problem that cannot be reduced further.

The existence of a (fast) simplification algorithm like this is a pretty good characterisation of problems that are paramaterisable[3].

[3] The formal term is FPT – Fixed-Parameter Tractable.

Kernelisation – example

Consider MAX-3-SAT (n variables, m clauses) where we want to find an assignment satisfying *at least* k clauses.

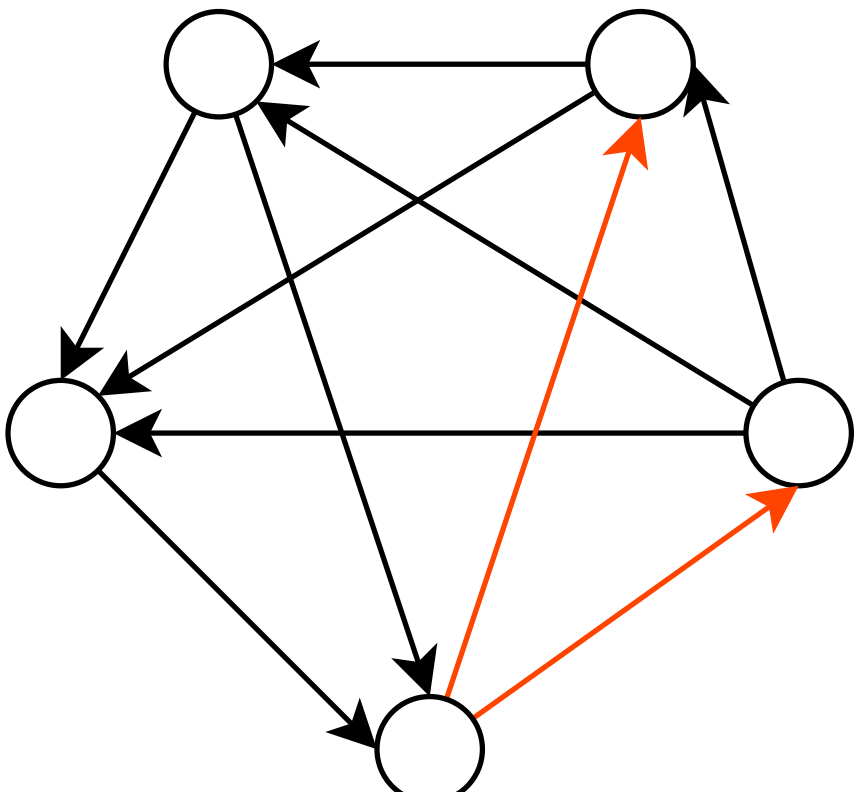
1. If $k \leq \frac{m}{2}$ then this is always possible – choose any assignment, if it satisfies fewer than half the clauses flip it.
2. Otherwise, $m < 2k$. Delete all variables not in any clause, getting $n \leq 6k$.

We thus obtain a kernel of size $\mathcal{O}(k)$.

Kernelisation – example 2

Dexter is organising a pawker tournament involving n cats. Every cat played with every other cat, and one always won. The results don't give a good final ranking though – there is no order of cats such that no cat lost any games to any cat lower in the ranking. Dexter is wondering if he could just invalidate some games to get a good ranking, but if he invalidates too many the cats will figure out the ruse. Help Dexter find at most k games which, when removed, would save the ranking!

Kernelisation – example 2



Given a tournament on n vertices, decide if it has a *feedback arc set* of size at most k (problem called FAST).

A feedback arc set is a set of edges A such that every cycle contains at least one edge from A. Equivalently, removing A from the graph makes it acyclic.

Kernelisation – example 2

The key observation is that if there is a cycle in a tournament then there necessarily is a triangle. Moreover:

Lemma: *A is an* inclusion-minimal *feedback arc set if and only if A is an inclusion minimal set of edges that, when reversed, give an acyclic graph.*

Proof: Right-to-left is immediate. For left-to-right, remove A and consider the topological order of the resulting DAG. We can then add all edges from A back – directly if it does not break the order, flipped if it does.

Kernelisation – example 2

The idea is to find edges that are involved in many triangles and flip them. Moreover, vertices outside of any triangles are boring. Kernelisation is thus:

1. *If an edge e is part of at least $k+1$ triangles, reverse e and reduce k by* 1.
2. *If a vertex v is not part of any triangle remove it from the graph.*

We need to show this preserves the equivalence.

Rule 1. is correct because if we didn't reverse e we would have to reverse $k+1$ other edges, one for each triangle, so the set would not be of size at most k.

Kernelisation – example 2

1. If an edge e is part of at least $k+1$ triangles, reverse e and reduce k by 1.
2. If a vertex v is not part of any triangle remove it from the graph.

Rule 2. is correct because removing v partitions the tournament into two sets – I, which had edges into v, and F, which had edges coming from v. This is a partition, since otherwise we'd get a triangle with v.

Clearly then, if we take the union of any feedback arc set in I and any feedback arc set in F, we get a feedback arc set on G; and any feedback arc set of G can be cleanly split into feedback arcs on I and F.

Kernelisation – example 2

Finally, if the tournament does contain a feedback arc set A of size at most k, then the result of this kernelisation has at most $k(k+2)$ vertices.

Every vertex of G is in a triangle, every triangle must contain an edge from A, and for every edge $e \in G$ there are at most k vertices in triangles containing e plus the two endpoints of e. So, k edges times $k+2$ vertices gives $k(k+2)$.

Kernelisation – example 2

We can apply our kernelisation and if the result has more than $k(k+2)$ vertices then the answer is NO. Otherwise, we can apply a brutal algorithm to solve the smaller instance.

In kernelisation we need to find all triangles, which takes $\mathcal{O}\left(|V|^3\right)$.

Kernelisation – example 2

Finally, to solve the problem on the reduced instance:

- find all triangles;
- if there are none, we are done;
- otherwise, if we've chosen k edges already the answer is NO;
- for each triangle choose one of the edges to be flipped and recurse (check all combinations).

At each step we choose one of three edges and we can recurse at most k times, therefore we get $\mathcal{O}\left(3^k(k(k+2))^2\right)$

This gives reasonably efficient algorithms for $k \leq 10$.

Next week

Next week there is no new topic.

We will talk about APD, show the grade thresholds, and be open for discussion.

See you next week

APD: 22.07.2025, 10:00 AM

Good luck!