Einführung und Definitionen

Caspar Nagy

27. Mai 2019

Gliederung

- Motivation
- Definitionen 1/2
- Beispiele für parametrisierte Probleme und ihre Komplexität
  - Bar Fight Prevention
    - Brute Force
    - Kernelization
    - Bounded Search Trees
  - Clique
    - mit k
    - mit Δ
- Definitionen 2/2
- Komplexitätsbeweise für parametrisierte Probleme
  - Theoretische Grundlagen
  - Clique für reguläre Graphen $\in W[1]$

Motivation – Wo wir bei TGI stehen geblieben sind

Viele Interessante Probleme $\in NP$

- Lösungen für Bar Fight Prevention (aka Vertex Cover) schon für $n = 1000$ sehr unhandlich
- Laufzeit kann drastisch reduziert werden, wenn wir den Lösungsraum einschränken

Frage:

- Welche Parameter vereinfachen unser Problem tatsächlich?
- Welche Laufzeit kann man mit Parametrisierung erreichen?

Definitionen 1/2

Parametrisiertes Problem

- Ein parametrisiertes Problem ist eine Sprache $L \subseteq \Sigma^* \times \mathbb{N}$. Bei Instanzen $(x, k) \in \Sigma^* \times \mathbb{N}$ bezeichnen wir k als *Parameter*.
- Die Größe eines parametrisierten Problems definieren wir als $|x| + k$

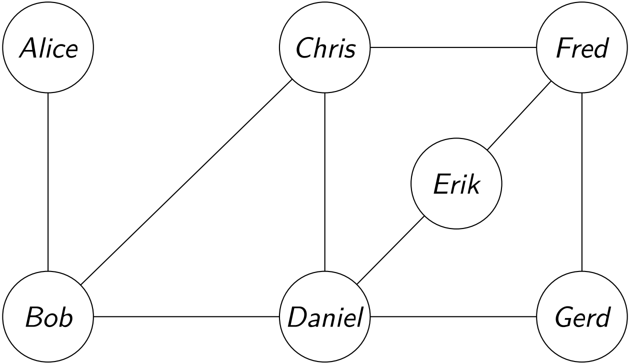
FPT (*Fixed Parameter Tractable*)

- Menge der parametrisierten Probleme, für die ein Algorithmus $\mathcal{A}$ existiert, der Instanzen in Zeit $\mathcal{O}(f(k) \cdot |(x, k)|^c)$ entscheidet.

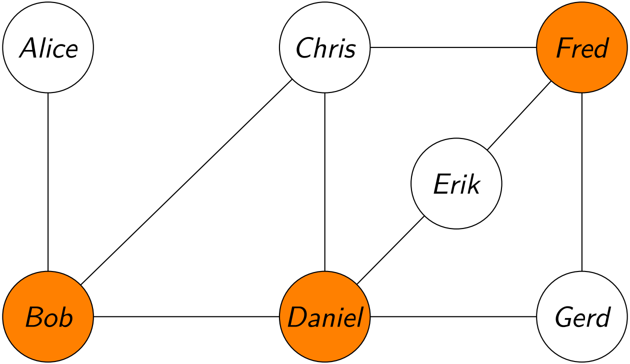
XP (*slice-wise polynomial*)

- Menge der parametrisierten Probleme, für die ein Algorithmus $\mathcal{A}$ existiert, der Instanzen in Zeit $\mathcal{O}(f(k) \cdot |(x, k)|^{g(k)})$ entscheidet.
  - f und g sind berechnenbar
  - f ist monoton wachsend

Beispiel Bar Fight Prevention



Beispiel Bar Fight Prevention



Beispiel – Bar Fight Prevention – Brute Force

```
min_size = INFINITY
for candidate in potenzmenge(g.V):
   if solution(candidate) and len(candidate) < min_size:
    best, min_size = candidate, len(candidate)
return best
```

- `solution(candidate)` wird 2^n mal aufgerufen
- sei $n = 1000$: $2^{1000} \approx 10^{301}$
  - terminiert nach Ende des Universums

Ansatz: Parameter k einführen, der Größe der Lösung beschränkt

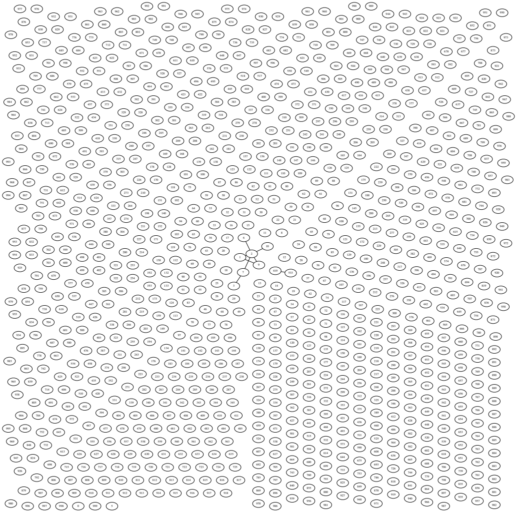
Beispiel – Bar Fight Prevention – Brute Force

```
for candidate in k_teilmengen(g.V, k):
   if solution(candidate):
    return candidate
return None
```

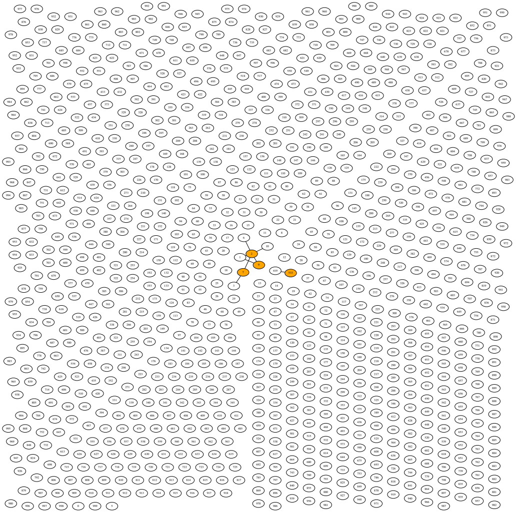
- ‘solution(candidate) wird $\binom{n}{k}$ mal aufgerufen
- sei $n = 1000, k = 10 : \binom{1000}{10} \approx 2,63 * 10^{23}$
  - terminiert nach einigen Jahrzehnten auf einem state-of-the-art Supercomputer

Noch immer unbefriedigend. Können wir den Suchraum weiter einschränken?

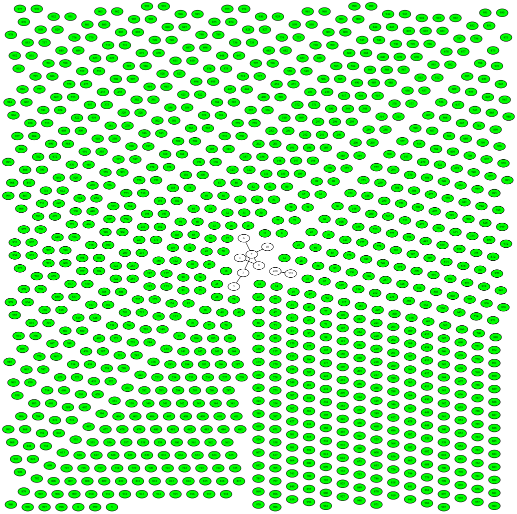
Beispiel – Bar Fight Prevention



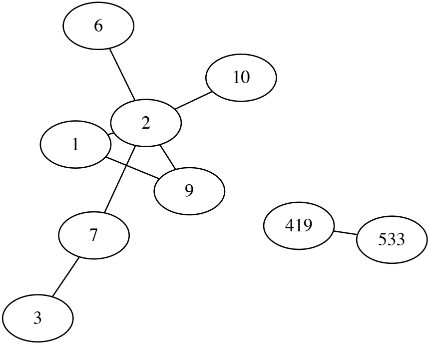
Beispiel – Bar Fight Prevention



Beispiel – Bar Fight Prevention

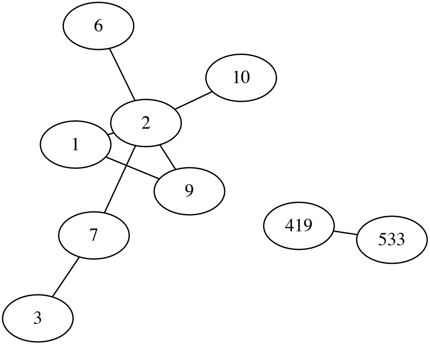


Beispiel – Bar Fight Prevention – Kernelization



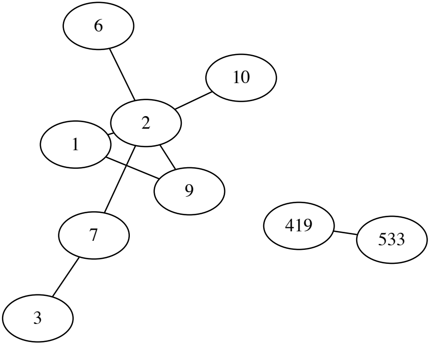
- Ist effizient berechnenbar
- Kann viele Instanzen wesentlich verkleinern
  - hilft uns aber nicht im worst-case
  - nutzt noch nicht unseren Parameter k

Beispiel – Bar Fight Prevention – Kernelization



sei $k = 4$

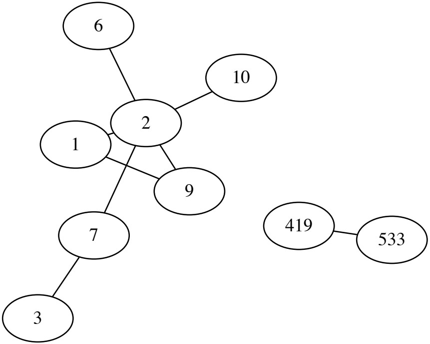
Beispiel – Bar Fight Prevention – Kernelization



sei $k = 4$

- Es ist egal ob wir 419 oder 533 herauswerfen.

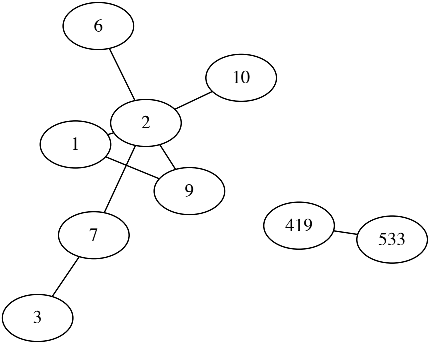
Beispiel – Bar Fight Prevention – Kernelization



sei $k = 4$

- Es ist egal ob wir 419 oder 533 herauswerfen.
- 2 muss immer herausgeworfen werden.

Beispiel – Bar Fight Prevention – Kernelization



Allgemein:

- Knoten mit *degree*$(n) = 1$ können wir hereinlassen, solange dadurch nicht direkt ein Konflikt entsteht.
- Knoten mit *degree*$(n) > k$ werden immer heraus geworfen.

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in \mathit{unknown} : \mathit{degree}(v) \leq k$

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks
    - $k = 10, \binom{200}{10} \approx 2,24 * 10^{16}$

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks
    - $k = 10, \binom{200}{10} \approx 2,24 * 10^{16}$
  - da außerdem:
    $\forall v \in unknown : degree(v) > 1$

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks
    - $k = 10, \binom{200}{10} \approx 2,24 * 10^{16}$
  - da außerdem:
    $\forall v \in unknown : degree(v) > 1$
    - im Worst Case (Kreis) 1 Mensch pro Konflikt

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks
    - $k = 10$, $\binom{200}{10} \approx 2,24 * 10^{16}$
  - da außerdem:
    $\forall v \in unknown : degree(v) > 1$
    - im Worst Case (Kreis) 1 Mensch pro Konflikt
    - also nur noch $\binom{k^2}{k}$ Checks

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

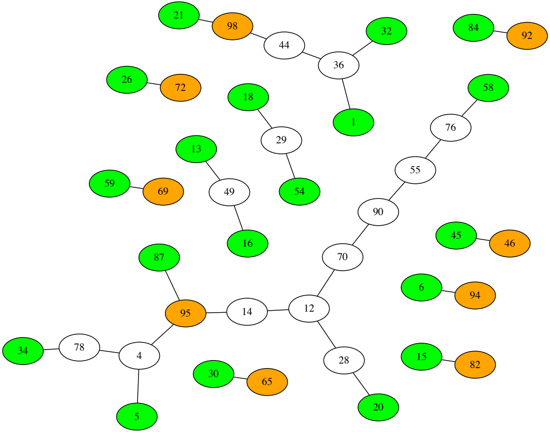
- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks
    - $k = 10, \binom{200}{10} \approx 2,24 * 10^{16}$
  - da außerdem:
    $\forall v \in unknown : degree(v) > 1$
    - im Worst Case (Kreis) 1 Mensch pro Konflikt
    - also nur noch $\binom{k^2}{k}$ Checks
    - $k = 10, \binom{100}{10} \approx 1,73 * 10^{13}$ Checks

Beispiel – Bar Fight Prevention – Kernelization

```
accepted = list()
denied = list()
unknown = list()
for v in G.V:
    if degree(v) == 0:
        accepted += v
    elif degree(v) == 1:
        if confict(v):
            rejected += v
            k -= 1
        else: accepted += v
    elif degree(v) > k:
        rejected += v
        k -= 1
    else:
        unknown += v
    if k < 0:
        fail()
if len(G.subplot(unknown + accepted).edges) > k*k:
    fail()
```

- Läuft in Polynomialzeit
- Was tun mit `unknown`?
  - $\forall v \in unknown : degree(v) \leq k$
  - $\Rightarrow$
    - $\sum_{v \in unknown} degree(v) \leq k^2$
    - $\leq 2 \cdot k^2$ Konfliktparteien
    - $\leq \binom{2k^2}{k}$ Checks
    - $k = 10, \binom{200}{10} \approx 2,24 * 10^{16}$
  - da außerdem:
    $\forall v \in unknown : degree(v) > 1$
    - im Worst Case (Kreis) 1 Mensch pro Konflikt
    - also nur noch $\binom{k^2}{k}$ Checks
    - $k = 10, \binom{100}{10} \approx 1,73 * 10^{13}$ Checks
    - $\Rightarrow$ einige Stunden auf einem Laptop

Beispiel – Bar Fight Prevention – Kernelization

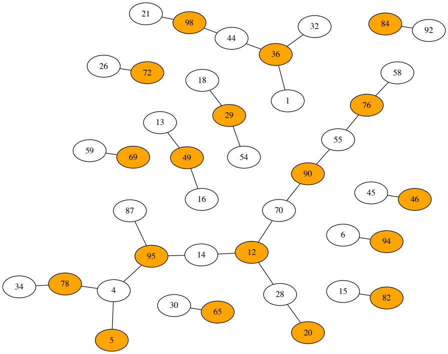


Beispiel – Bar Fight Prevention – Kernelization

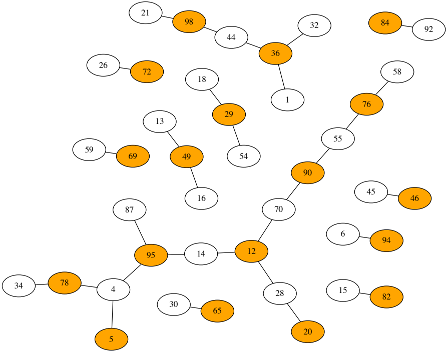
Fazit

- Kernelization verkleinert bestimmte Instanzen, teilweise drastisch
- Kernelization mit Parameter kann viele worst-case-Instanzen abfangen und dadurch die Laufzeit begrenzen
- Fortgeschrittene Techniken im nächsten Vortrag

Beispiel – Bar Fight Prevention – Weitere Ansätze

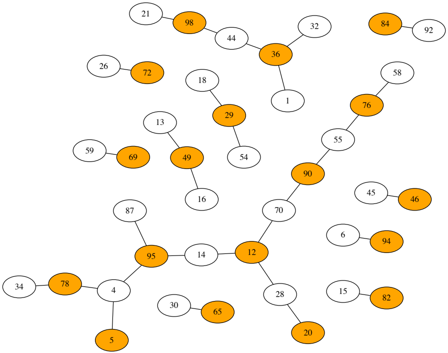


Beispiel – Bar Fight Prevention – Weitere Ansätze



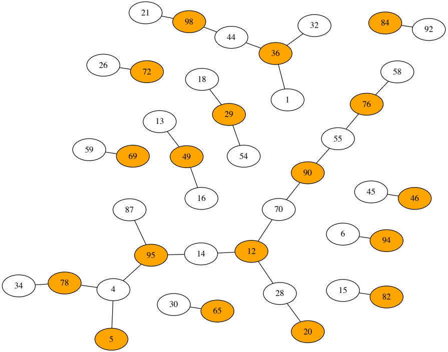
- Jeder Konflikt muss gelöst werden.

Beispiel – Bar Fight Prevention – Weitere Ansätze



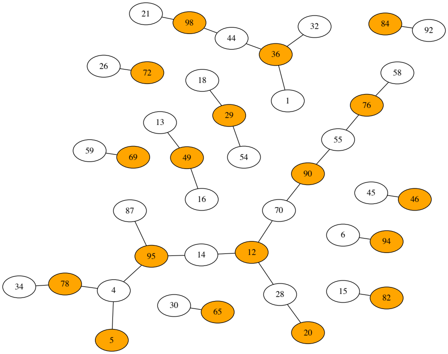
- Jeder Konflikt muss gelöst werden.
- Normalerweise $2^{|E|}$ Checks

Beispiel – Bar Fight Prevention – Weitere Ansätze



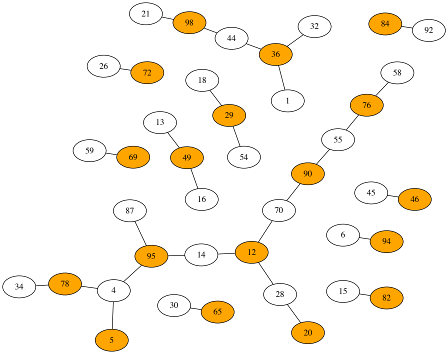
- Jeder Konflikt muss gelöst werden.
- Normalerweise $2^{|E|}$ Checks
- Da aber jeder Konflikt gelöst werden muss und die Lösung nur k Knoten enthalten darf:

Beispiel – Bar Fight Prevention – Weitere Ansätze



- Jeder Konflikt muss gelöst werden.
- Normalerweise $2^{|E|}$ Checks
- Da aber jeder Konflikt gelöst werden muss und die Lösung nur k Knoten enthalten darf:
  - Abbruch nach k Rekursionen

Beispiel – Bar Fight Prevention – Weitere Ansätze



- Jeder Konflikt muss gelöst werden.
- Normalerweise $2^{|E|}$ Checks
- Da aber jeder Konflikt gelöst werden muss und die Lösung nur k Knoten enthalten darf:
  - Abbruch nach k Rekursionen
  - Also nur 2^k Checks

Beispiel - Bar Fight Prevention – Bounded Search Trees

```
def bounded_search(G, k, solution=[]):
    if k == 0:
        return []
    v1, v2 = G.random_edge()
    for v in (v1, v2):
        g = G.copy()
        g.remove_node(v)
        if len(g.edges()) == 0:
            return solution + [v]
    g1 = G.copy()
    g2 = G.copy()
    g1.remove_node(v1)
    g2.remove_node(v2)
    return max(
        bounded_search(g1, k-1, solution + [v1]),
        bounded_search(g2, k-1, solution + [v2])
    )
```

Beispiel - Bar Fight Prevention - Fazit

- exponentieller Worst-case ohne Parametrisierung
- Parametrisierung + Brute Force polynomiell mit Laufzeit $\binom{n}{k} \in \mathcal{O}(n^k)$
  - jedoch nur praktikabel für kleine k
- Parametrisierung + Brute Force + Kernelization in $\binom{k^2}{k} * n^{\mathcal{O}(1)}$
- Bounded Search Trees in $2^k * k * n^{\mathcal{O}(1)}$

Beispiel - Clique

Gesucht ist eine paarweise verbundene Teilknotenmenge der Größe k

- Naiver $\binom{n}{k} \in \mathcal{O}(n^k)$ Algorithmus
  - Testen aller k-Teilmengen
  - $\in XP$
    - hoffnungslos für große k

Finden wir einen Algorithmus ohne k im Exponent?

Beispiel - Clique

Gesucht ist eine paarweise verbundene Teilknotenmenge der Größe k

- Naiver $\binom{n}{k} \in \mathcal{O}(n^k)$ Algorithmus
  - Testen aller k-Teilmengen
  - $\in XP$
    - hoffnungslos für große k

Finden wir einen Algorithmus ohne k im Exponent?

Vielleicht schon, wahrscheinlich aber nicht.

Beispiel - Clique – anderer Parameter

Betrachten wir nun Clique für Graphen mit maximalem Grad Δ

Algorithmus:

- iteriere über alle n Knoten
  - Checke alle Teilmengen der Adjazenten Knoten
    - das sind höchstens 2^{Δ}
  - Δ^2 Operationen pro Check
- Laufzeit von $\mathcal{O}(2^{\Delta} * \Delta^2 * n)$

Beispiele – Round Up

Problem	Vorgestelle Laufzeit
Bar Fight Prevention – Kernelization	$\mathcal{O}(\binom{k^2}{k} \cdot n)$
Bar Fight Prevention – B. Search Trees	$\mathcal{O}(2^k \cdot k \cdot n)$
Clique mit k	$n^{\mathcal{O}(k)}$
Clique mit Δ	$\mathcal{O}(2^\Delta \cdot \Delta^2 \cdot n)$

Beispiele – Round Up

- Parametrisierung ist eine Kunst
- Viele mögliche Parameter pro Problem
- Je mehr Algorithmentechniken man zur Verfügung hat, desto besser
- Obwohl XP und FPT ähnlich sind, ist der Unterschied in der Praxis enorm.
- Scheinbar gibt es parametrisierte Probleme in $XP \setminus FPT$
  - Wie erkennen wir sie?

Definitionen 2/2

Aus TGI kennen wir die Mengen *P* und *NP*. Für parametrisierte Probleme gibt es analog *FPT*/*XP* und *W[1]*

- W[1] ist die Menge aller parametrisierten Probleme, die mindestens so komplex sind wie das Finden einer CLIQUE der Größe k.
- Analog zu NP wird die W[1]-Schwere über polynomielle Reduktion gezeigt.
- Das alles ist natürlich sinnlos, sollte $P = NP$ oder CLIQUE $\in$ *FPT* sein.

Komplexitätsbeweise - Polynomielle Reduktion

Definition

Seien $A, B \subseteq \Sigma^* \times \mathbb{N}$ zwei parametrisierte Probleme. Eine *parametrisierte Reduktion* von A nach B ist ein Algorithmus, der gegeben einer Instanz (x, k) von A eine Instanz (x', k') von B ausgibt, sodass:

1. $(x, k) \in A \iff (x', k') \in B$
2. $k' \leq g(k)$
3. die Laufzeit der Reduktion ist $\in \mathcal{O}(f(k) \cdot |x|^{\mathcal{O}(1)})$
    - f und g sind berechnenbare, monoton steigende Funktionen

Komplexitätsbeweise - Polynomielle Reduktion

Theorem 13.2

Wenn $B \in FPT$ und eine *parametrisierte Reduktion* von A auf B existiert, so ist auch $A \in FTP$

Komplexitätsbeweise - Polynomielle Reduktion

Theorem 13.4: Es existiert eine *parametrisierte Reduktion* von $\textsc{Clique}$ auf $\textsc{Clique}$ *auf regulären Graphen*.

Beweis: Gegeben sei eine Instanz (G, k) von $\textsc{Clique}$. Für die trivialen Fälle $k \leq 2$ geben wir eine triviale Instanz zurück. Ansonsten verfahren wir wie folgt:

- sei $d := \max_{v \in G.V} \mathit{degree}(v)$
- Erzeuge G' als d Kopien von G
- Erzeuge für jeden Knoten v in G $d - \mathit{degree}(v)$ Hilfsknoten und verbinde sie mit jeder Kopie von v in G'

Korrektheit:

0. Die Regularität des Graphen folgt direkt aus der Konstruktion
1. Da die Kopien nicht direkt verbunden sind, entstehen keine neuen Cliquen $(k > 2)$, alte bleiben erhalten.
2. wähle $g(k) := \mathit{id}$
3. Laufzeit von k unabhängig, offensichtlich polynomiell in n

Fazit

Fragen?