Karlsruher Institut für Technologie
Institut für Theoretische Informatik

Prof. Dr. Peter Sanders
Moritz Kobitzsch, Dennis Schieferdecker

6. Übungsblatt zu Algorithmen II im WS 2011/2012

http://algo2.iti.kit.edu/AlgorithmenII.php
{kobitzsch,sanders,schieferdecker}@kit.edu

Musterlösungen

Aufgabe 1 (*Kleinaufgaben: Laufzeiten*)

a) Sei $T(n, \varepsilon)$ die Laufzeit eines Approximationsalgorithmus und $g(n, \varepsilon)$ seine Approximationsgarantie. Geben Sie für die folgenden Fälle an, ob der Algorithmus ein PTAS, FPTAS oder keines von beiden ist. Begründen Sie Ihre Antwort jeweils kurz.

- $T_1(n, \varepsilon) = \frac{1}{\varepsilon} \cdot (4n^3 + n^2)$, $g_1(n, \varepsilon) = (1 - \varepsilon)$
- $T_2(n, \varepsilon) = \frac{1}{\varepsilon} \cdot n^2$, $g_2(n, \varepsilon) = (1 + 2\varepsilon)$
- $T_3(n, \varepsilon) = \sqrt{n} + n^{\frac{3}{2}}$, $g_3(n, \varepsilon) = 2 + \frac{1}{n}$
- $T_4(n, \varepsilon) = n \cdot \log \frac{1}{\varepsilon}$, $g_4(n, \varepsilon) = (1 - \varepsilon)$
- $T_5(n, \varepsilon) = \varepsilon + e^{\log n} + n^5$, $g_5(n, \varepsilon) = (1 + \varepsilon)$
- $T_6(n, \varepsilon) = n^{\frac{1}{\varepsilon}} + n^5$, $g_6(n, \varepsilon) = (2 + \varepsilon)$

b) Sei $f(n, k)$ die Laufzeit eines Algorithmus mit n der Eingabegröße des Problems und k ein beliebiger Parameter. Geben Sie an welche der folgenden Laufzeiten ein Problem *fixed-parameter-tractable* machen. Begründen Sie Ihre Antwort jeweils kurz.

- $f_1(n, k) = 3k^2n^2$
- $f_2(n, k) = n^k \cdot k^2 \cdot \sqrt{n^e}$
- $f_3(n, k) = 3n^2 + 2nk$
- $f_4(n, k) = e^k \cdot \sqrt{n}$
- $f_5(n, k) = e^n \cdot \sqrt{k}$
- $f_6(n, k) = k^3 \cdot \log n^2$

c) (*) Gegeben seien folgende Rekurrenzrelationen zur Beschreibung von Laufzeiten. Bestimmen Sie (z.B. mit Hilfe erzeugender Polynome) das asymptotische Wachstum dieser Laufzeiten.

- $T_1(n, k) = 2T_1(n, k - 3) + T_1(n, k - 6)$
- $T_2(n, k) = T_2(n, k - 1) + T_2(n, k - 4) + n$
- $T_3(n, k) = 9T_3(n, k - 3) - 4T_3(n, k - 1)$
- $T_4(n, k) = T_4(n, k - 4) + T_4(n, k - 2) + n^2$

Musterlösung:

a) Ein Approximationsalgorithmus wird als PTAS bezeichnet, wenn seine Laufzeit $T(n, \varepsilon)$ polynomiell in n ist und sich seine Approximationsgarantie beliebig nahe der 1 nähern kann. Für ein FPTAS muss zusätzlich $T(n, \varepsilon)$ polynomiell in $\frac{1}{\varepsilon}$ sein. Mit dieser Definition ergibt sich für die angegebenen Algorithmen:

- $T_1(n, \varepsilon) = \frac{1}{\varepsilon} \cdot (4n^3 + n^2)$, $\quad g_1(n, \varepsilon) = (1 - \varepsilon)$
  Bei dem angegebenen Algorithmus handelt es sich um ein FPTAS. $T_1(n, \varepsilon)$ hängt polynomiell von n und $\frac{1}{\varepsilon}$ ab und die Approximationsgarantie kann beliebig nahe an die 1 herankommen.
- $T_2(n, \varepsilon) = \frac{1}{\varepsilon} \cdot n^2$, $\quad g_2(n, \varepsilon) = (1 + 2\varepsilon)$
  Bei dem angegebenen Algorithmus handelt es sich um ein FPTAS. Es gilt die gleiche Begründung wie bei $T_1(n, \varepsilon)$ und $g_1(n, \varepsilon)$. Will man die klassische Approximationsgarantie ohne den Faktor 2 sehen, substituiert man einfach $\delta = 2\varepsilon$.
- $T_3(n, \varepsilon) = \sqrt{n} + n^{\frac{3}{2}}$, $\quad g_3(n, \varepsilon) = 2 + \frac{1}{n}$
  Bei dem angegebenen Algorithmus handelt es sich weder um ein FPTAS noch um ein PTAS. Die Approximationsgarantie hängt von n ab und kann nicht beliebig nahe an 1 herankommen.
- $T_4(n, \varepsilon) = n \cdot \log \frac{1}{\varepsilon}$, $\quad g_4(n, \varepsilon) = (1 - \varepsilon)$
  Bei dem angegebenen Algorithmus handelt es sich um ein FPTAS. Die Approximationsgarantie kann sich beliebig der 1 nähern. $T_4(n, \varepsilon)$ hängt polynomiell von n und auch von $\frac{1}{\varepsilon}$ (da $\log x = O(n) = O(poly(n))$).
- $T_5(n, \varepsilon) = \varepsilon + e^{\log n} + n^5$, $\quad g_5(n, \varepsilon) = (1 + \varepsilon)$
  Bei dem angegebenen Algorithmus handelt es sich um ein FPTAS. $T_5(n, \varepsilon)$ ist polynomiell in n ($e^{\log n} = n$) und $\frac{1}{\varepsilon}$ ($\varepsilon = \frac{1}{\varepsilon}^{-1}$). Außerdem kann sich die Approximationsgarantie beliebig der 1 nähern.
- $T_6(n, \varepsilon) = n^{\frac{1}{\varepsilon}} + n^5$, $\quad g_6(n, \varepsilon) = (2 + \varepsilon)$
  Bei dem angegebenen Algorithmus handelt es sich um ein PTAS. Die Approximationsgarantie kann sich beliebig der 1 nähern und $T_6(n, \varepsilon)$ hängt zwar polynomiell von n ab, aber nicht von $\frac{1}{\varepsilon}$. Für die klassische Darstellung der Approximationsgarantie ersetzt man $\varepsilon = \delta - 1$.

Musterlösung:

b) Ein Problem heisst *fixed parameter tractable*, falls ein Algorithmus zu dessen Lösung existiert, dessen Laufzeit durch $O(f(k) \cdot poly(n))$ mit $poly(n)$ Polynom nur in n und, $f(k)$ beliebige Funktion nur in k abschätzbar ist. Damit ergibt sich für die angegebenen Probleme:

- $f_1(n,k) = 3k^2n^2$
  Das beschriebene Problem ist *fixed parameter tractable*. $f_1(n,k)$ ist polynomiell in n (n^2) und davon unabhängig abhängig in k (k^2).
- $f_2(n,k) = n^k \cdot k^2 \cdot \sqrt{n^e}$
  Das beschriebene Problem ist nicht *fixed parameter tractable*. $f_2(n,k)$ ist durch $n^k \cdot \sqrt{n^e}$ zwar polynomiell in n, aber dies ist nicht unabhängig von k.
- $f_3(n,k) = 3n^2 + 2nk$
  Das beschriebene Problem ist *fixed parameter tractable*. Durch $poly(n) = n^2$ und $f(k) = k$ lässt sich $f_3(n,k)$ abschätzen ($3n^2 + 2nk = O(n^2k)$.
- $f_4(n,k) = e^k \cdot \sqrt{n}$
  Das beschriebene Problem ist *fixed parameter tractable*. $f_4(n,k)$ ist polynomiell in n ($\sqrt{n}$) und davon unabhängig abhängig von k (e^k).
- $f_5(n,k) = e^n \cdot \sqrt{k}$
  Das beschriebene Problem ist nicht *fixed parameter tractable*. $f_5(n,k)$ ist nicht polynomiell in n (e^n).
- $f_6(n,k) = k^3 \cdot \log n^2$
  Das beschriebene Problem ist *fixed parameter tractable*, da $f_6(n,k)$ polynomiell in n ist (da $\log n^2 = O(n) = O(poly(n))$) und davon unabhängig von k abhängig (k^3).

Musterlösung:

c)

- $T_1(n,k) = 2T_1(n,k-3) + T_1(n,k-6)$
  Verzweigungsvektor: $\langle 3,3,6\rangle$
  Nach Tabelle ergibt sich $T_1(n,k) = O(1.342^k)$. Beachte, dass in diesem Fall die Laufzeit unabhängig von n ist!
- $T_2(n,k) = T_2(n,k-1) + T_2(n,k-4) + n$
  Verzweigungsvektor: $\langle 1,4\rangle$
  Nach Tabelle ergibt sich ein Ausführungsbaum der Größe $O(1.380^k)$. Da in jedem Knoten noch n Arbeit erledigt werden muss, ist die Laufzeit $T_2(n,k) = O(1.380^k n)$.
- $T_3(n,k) = 9T_3(n,k-3) - 4T_3(n,k-1)$
  Verzweigungsvektor: Kann nicht angegeben werden.

  Per Hand nachgerechnet:
  Das erzeugende Polynom $x^k = 9x^{k-3} - 4x^{k-1}$ nach x aufgelöst, ergibt:

$$\begin{aligned} & x^k = 9x^{k-3} - 4x^{k-1} \\ \Leftrightarrow\ & x^3 = 9 - 4x^2 \\ \Leftrightarrow\ & 0 = (x^2 + x - 3)(x+3) \\ \Leftrightarrow\ & 0 = (x + \frac{1+\sqrt{13}}{2})(x - \frac{1+\sqrt{13}}{2})(x+3) \end{aligned}$$

  Die größte Lösung der Gleichung ist $x = \frac{1+\sqrt{13}}{2} \approx 2.303$. Damit ergäbe sich eine Laufzeit von $T_3(n,k) = O(2.303^k)$. Allerdings ist $x = -3$ die *betragsmäßig* größte Lösung. Die tatsächliche Laufzeit ist also $T_3(n,k) = O(3^k)$.
  Bemerkung: Das negative Vorzeichen der Basis wird durch das O-Kalkül geschluckt!
- $T_4(n,k) = T_4(n,k-4) + T_4(n,k-2) + n^2$
  Verzweigungsvektor: $\langle 2,4\rangle$ – nicht in Tabelle.
  Trick: Verwende die Wurzel der Basis von $\langle \frac{2}{2}, \frac{4}{2}\rangle = \langle 1,2\rangle$. Damit ergibt sich eine Laufzeit von $T_4(n,k) = O(1.272^k)$.

  Nachgerechnet:
  Das erzeugende Polynom $x^k = x^{k-4} + x^{k-2}$ nach x aufgelöst, ergibt:

$$\begin{aligned} & x^k = x^{k-4} + x^{k-2} \\ \Leftrightarrow\ & x^4 = 1 + x^2 \\ \overset{y:=x^2}{\Leftrightarrow}\ & y^2 = 1 + y \\ \Leftrightarrow\ & y_{1,2} = \frac{1 \pm \sqrt{1+4}}{2} \end{aligned}$$

  Verwende die größere Lösung $y = \frac{1+\sqrt{5}}{2} \approx 1.618$ und bestimme damit $x = \sqrt{\frac{1+\sqrt{5}}{2}} \approx 1.272$ (auch hier ist nur die größere der beiden möglichen Lösungen wichtig). Damit hat der Ausführungsbaum eine Größe von $O(1.272^k)$. Da in jedem Knoten noch n^2 Arbeit erledigt werden muss, ist die Laufzeit $T_4(n,k) = O(1.272^k n^2)$.

Aufgabe 2 (*Analyse+Rechnen: Vertex-Cover*)

Gegeben sei folgender Algorithmus zur Berechnung eines *vertex cover* C für einen Graph $G = (V, E)$:

1. Initialisiere die Ergebnismenge $C = \varnothing$ als leere Menge.
2. Wähle Kante $(u, v) \in E$ beliebig.
3. Füge u, v zu C hinzu.
4. Entferne u, v und alle inzidenten Kanten aus G.
5. Wiederhole solange G noch Kanten hat

Nach Abschluss des Algorithmus ist $C \subseteq V$ ein *vertex cover*, d.h. für jede Kante $(u, v) \in E$ ist einer ihrer beiden Knoten in C. Falls o.b.d.A. $u \in C$ sagt man auch Knoten u *überdeckt* Kante (u, v).

a) Zeigen Sie, dass der angegebene Algorithmus ein korrektes *vertex cover* berechnet.

b) Geben Sie ein Beispiel an, in dem der Algorithmus ein minimales *vertex cover* liefert.

c) Geben Sie ein Beispiel an, in dem der Algorithmus kein minimales *vertex cover* liefert.

d) Zeigen oder widerlegen Sie, dass der Algorithmus eine 2-Approximation für *vertex cover* berechnet, d.h. dass er höchstens doppelt soviele Knoten auswählt als minimal nötig.

Betrachten Sie abschließend diesen alternativen Algorithmus zur Bestimmung einer 2-Approximation von *vertex cover*:

1. Initialisiere die Ergebnismenge $C = \varnothing$ als leere Menge.
2. Wähle Knoten $u \in V$ mit minimalem Grad.
3. Füge u zu C hinzu.
4. Entferne u und alle inzidenten Kanten aus G
5. Wiederhole solange G noch Kanten hat

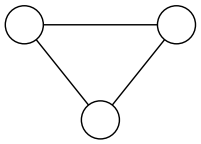
Der Algorithmus berechnet offenbar –mit ähnlichen Argumenten wie in Teilaufgabe (a)– ein *vertex cover*. Es bleibt folgende Frage zu beantworten:

e) Zeigen oder widerlegen Sie, dass der Algorithmus eine 2-Approximation für *vertex cover* berechnet, d.h. dass er höchstens doppelt soviele Knoten auswählt als minimal nötig.

Musterlösung:

a) Nach Abschluss enthält der Graph keine Kanten mehr. Da eine Kante nur entfernt wird, wenn einer ihrer Knoten in C aufgenommen wurde, ist folglich jede Kante $e \in E$ von einem Knoten überdeckt. Damit ist C ein *vertex cover*.

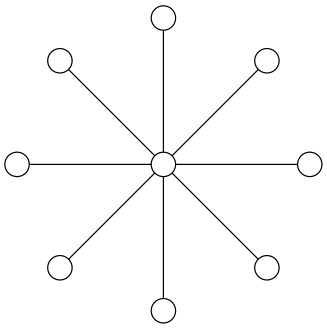
b) In folgendem Graphen werden immer genau zwei Knoten ausgewählt. Dies ist optimal, da einzelner Knoten nur zwei der drei Kanten überdecken.



c) In folgendem Graphen werden immer zwei Knoten ausgewählt. Das ist nicht optimal, da ein einzelner Knoten genügen würde.

d) Sei A die Menge der in Schritt 2 ausgewählten Kanten. Es gilt $|C| = 2|A|$, da beide Knoten jeder ausgewählten Kante zu C hinzugefügt und anschließend zusammen mit allen inzidenten Kanten aus G entfernt werden, so dass sie nicht noch einmal ausgewählt werden können. Damit folgt auch, dass keine zwei Kanten aus A einen Knoten gemeinsam haben können. Sei nun C^* ein minimales *vertex cover*. C^* enthält nach Definition mindestes einen Knoten jeder Kante, also insbesondere einen Knoten jeder Kante aus A. Da keine zwei Kanten aus A vom gleichen Knoten aus C^* überdeckt werden können, gilt $|C^*| \geq |A|$. Es folgt $|C| = 2|A| \leq 2|C^*|$. Die Lösung C des angegebenen Algorithmus ist also maximal doppelt so groß wie die optimale Lösung C^*. Damit berechnet der Algorithmus eine 2-Approximation.

e) Der Algorithmus berechnet keine 2-Approximation. Folgender Graph ist ein Gegenbeispiel:



Ein optimales *vertex cover* benötigt nur den Knoten in der Mitte. Der angegebene Algorithmus markiert allerdings $n - 1$ Knoten (entweder alle Randknoten, oder den mittleren Knoten und alle Randknoten bis auf einen).

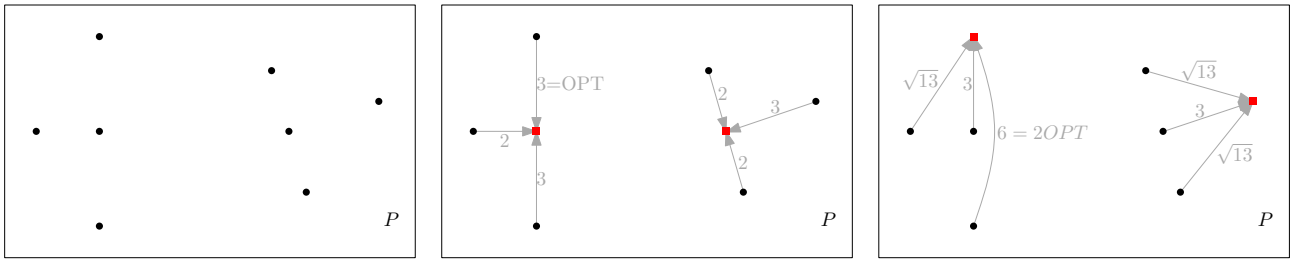
Aufgabe 3 (*Analyse: Metrisches k-Zentren Problem (*)*)

Gegeben sei eine Menge an Punkten $P \subset \mathbb{R}^2$ in der Ebene sowie eine Zahl $k > 0$. Gesucht ist eine k-elementige Teilmenge $K \subset P$ dieser Punkte, genannt Zentren, so dass für jeden Punkt $p \in P$ der maximale Abstand zu seinem nächstgelegenen Zentrum minimal ist.

Es existiert folgender *greedy* Algorithmus, der eine 2-Approximation des Problems berechnet:

1. Wähle beliebigen Punkt aus P als erstes Zentrum
2. Wähle Punkt aus P als nächstes Zentrum mit größter Entfernung zu allen bisherigen Zentren (d.h. der den maximalen kürzesten Abstand zu einem Zentrum besitzt)
3. Wiederhole bis k Zentren gewählt worden sind

Das folgende Beispiel veranschaulicht die Problemstellung für $k = 2$:



Links ist eine Punktmenge P abgebildet. In der Mitte ist eine optimale Lösung zu sehen. Die roten Quadrate sind die ausgewählten Zentren. Die Kanten geben das nächstgelegene Zentrum für jeden Knoten sowie den Abstand an. Rechts ist eine weitere aber suboptimale Lösung aufgezeigt.

Zunächst einige allgemeine Fragen zu diesem Algorithmus:

a) Beschreiben Sie in Worten, welche Bedeutung OPT sowie die Aussage eine Lösung sei eine 2-Approximation des metrischen k-Zentren Problems, haben.

b) Handelt es sich bei dem angegebenen Algorithmus um ein PTAS, ein FPTAS oder um keines von beiden. Begründen Sie kurz.

Im Folgenden soll gezeigt werdem, dass der Algorithmus tatsächlich eine 2-Approximation berechnet. Dafür sind zunächst einige Vorüberlegungen nötig.

c) Zeigen Sie, bei einer Auswahl von $k + 1$ Punkten aus P existieren immer mindestens 2 Punkte, die das gleiche nächstgelegene Zentrum haben.

d) Gegeben eine optimale Lösung, wie groß kann der Abstand zwischen zwei Punkten maximal sein, wenn diese das gleiche nächstgelegene Zentrum besitzen?

e) In einer Lösung des *greedy* Algorithmus sei der maximale Abstand eines Punktes $p \notin K$ zu seinem nächstgelegenen Zentrum $> l$. Zeigen Sie, dass l eine untere Schranke für den Abstand zwischen je zwei der Zentren $k_i,\ k_j \in K,\ i \neq j$ der Lösung darstellt.

f) Zeigen Sie mit obigen Aussagen, dass der angegebene *greedy* Algorithmus eine 2-Approximation für das Problem berechnet. Nehmen Sie dazu an, in der Lösung des Algorithmus sei der maximale Abstand eines Punktes $p \notin K$ zu seinem nächstgelegenen Zentrum $> 2 \cdot OPT$, und führen Sie diese Aussage zum Widerspruch.

Hinweis: Machen Sie zunächst eine Aussage über die paarweisen Abstände von $k + 1$ speziell gewählten Punkten. Verwenden Sie anschließend einen Vergleich zu Abständen in der optimalen Lösung, um zum Widerspruch zu gelangen.

Musterlösung:

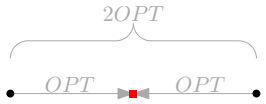
a) Im metrischen k-Zentren Problem charakterisiert OPT den maximalen Abstand eines Punktes zu seinem nächstgelegenen Zentrum. Eine 2-Approximation bedeutet, dass der Abstand eines Punktes zu seinem nächstgelegenen Zentrum höchstens doppelt so groß ist wie OPT (das rechte Bild in der Aufgabenstellung beschreibt eine 2-Approximation).

b) Der beschriebene *greedy* Algorithmus ist weder ein PTAS noch ein FPTAS, da sich seine Approximationsgüte nicht beliebig der 1 annähern lässt (bei polynomieller Laufzeit ist der Algorithmus immerhin in APX).

c) Angenommen k Punkte haben paarweise verschiedene nächstgelegene Zentren. Damit ist jeder Punkt einem anderen Zentrum zugeordnet und alle Zentren sind verwendet. Ein weiterer Punkt hätte auf alle Fälle ein schon verwendetes Zentrum als nächstgelegenes Zentrum. Wäre dies nicht der Fall, so gäbe es mehr als k Zentren oder die bisherigen Punkte hätten nicht alle paarweise verschiedene nächstgelegene Zentren.

   Dieses Prinzip wird auch *pigeon hole principle* genannt.

d) Wie in der Abbildung zu sehen, können sich beide Punkte auf entgegengesetzten Seiten des Zentrums befinden mit maximalem Abstand OPT. Damit ist ihr Abstand zueinander $2 \cdot OPT$.



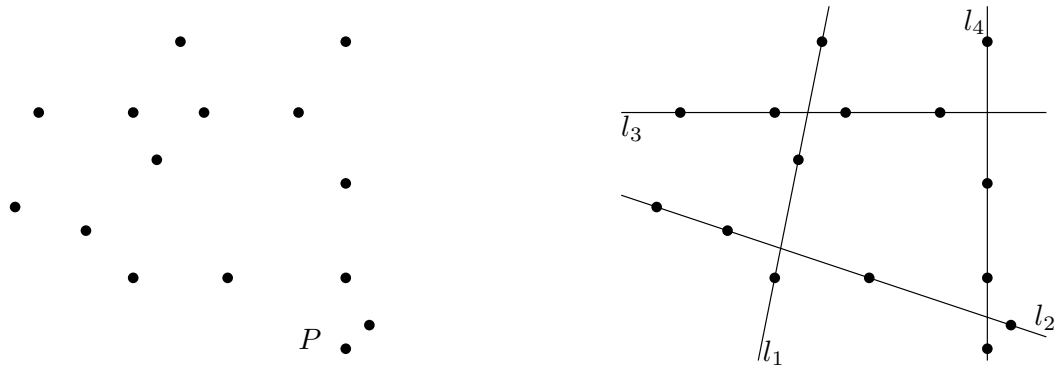
   (Hinweis: Für diese Aussage wurde der metrische Raum benötigt!)

e) Nach Voraussetzung ist Abstand $d(p, k) > l$ f.a. $k \in K$ und damit auch f.a. $k \in K/\{k_j\}$. Angenommen es gelte für einen Abstand $d(k_i, k_j) \leq l$. O.b.d.A. werde k_i vor k_j als Zentrum ausgwählt. Dann würde im weitern Verlauf des Algorithmus p anstatt k_j als Zentrum gewählt werden, da p den größeren minimalen Abstand zu den bisherigen Zentren hat. Da aber k_j ein Zentrum ist, muss $d(k_i, k_j) > l$ gelten.

f) Angenommen, in der Lösung des Algorithmus sei der maximale Abstand eines Punktes $p \notin K$ zu seinem nächstgelegenen Zentrum $> 2 \cdot OPT$. Nach Teilaufgabe (e) wäre der Abstand zwischen allen Zentren $> 2 \cdot OPT$. Das würde bedeuten, es gäbe $k + 1$ Punkte mit einem paarweisen Abstand $> 2 \cdot OPT$ (Punkt p sowie die k Zentren). Wähle zwei dieser Punkte, die in einer optimalen Lösung das gleiche nächste Zentrum haben. Teilaufgabe (c) belegt die Existenz dieser Punkte. Nach Teilaufgabe (d) hätten sie allerdings einen Abstand $\leq 2 \cdot OPT$. Widerspruch zu der Aussage, dass alle diese Punkte einen paarweisen Abstand $> 2 \cdot OPT$ besitzen.

Aufgabe 4 (*Analyse:* line shooting *Problem*)

Gegeben seien n Punkte $P \subset \mathbb{R}^2$ in der Ebene sowie eine Zahl $k > 0$. Das *line shooting* Problem besteht darin zu bestimmen, ob es eine Menge L von k Geraden gibt, so dass jeder Punkt in P von mindestens einer dieser Geraden getroffen wird. Eine Probleminstanz wird durch das Tupel (P, k) charakterisiert.
In den Bildern sehen Sie links eine Punktmenge P und rechts eine mögliche Lösung für $(P, 4)$.



a) Begründen Sie kurz, warum eine Gerade l, die mehr als k Punkte trifft, Teil einer Lösung für die Probleminstanz (P, k) sein muss bei beliebigem P.

b) Begründen Sie kurz, warum die Instanz $(P, 3)$ des *line shooting* Problems keine Lösung besitzt mit P wie in obiger Abbildung.

c) Geben Sie einen Algorithmus an, der eine Instanz (P, k) des *line shooting* Problems exakt löst für beliebiges P. Bauen Sie dazu einen Suchbaum mit beschränkter Tiefe auf, der alle Kombination von k Geraden, die jeweils mindestens zwei Punkte treffen, generiert. Die Suchbaumtiefe und der Verzweigungsgrad sollen dabei polynomiell in k, der Aufwand pro Knoten polynomiell in $n = |P|$ sein.

   Hinweise: Verwalten Sie in jedem Knoten des Suchbaumes k Einträge, die jeweils bis zu zwei Punkte halten können (und damit eine Gerade definieren). Ein Suchbaum der Höhe $O(k)$ genügt.

d) Zeigen Sie, dass das *line shooting* Problem *fixed parameter tractable* bezüglich k ist. Geben Sie dazu die asymptotische Laufzeit Ihres Algorithmus in Abhängigkeit von n und k an.

Musterlösung:

a) Allgemein gilt, dass eine Gerade, die $K > k$ Punkte trifft, Teil einer Lösung für (P, k) sein muss. Wäre diese Gerade nicht Teil der Lösung, so würde man K andere Geraden benötigen, um diese Punkte zu treffen. Da $K > k$ wäre dies für eine Lösung von (P, k) nicht zugelassen.

b) In der rechten Abbildung sieht man drei Geraden, die jeweils 4 Punkte treffen (l_2, l_3, l_4). Diese müssen nach obiger Begründung Teil einer Lösung von $(P, 3)$ sein. Da diese Geraden aber nicht ausreichen, um alle Punkte von P treffen, existiert keine Lösung für $(P, 3)$.

c) Idee: Der Algorithmus baut systematisch alle Möglichkeiten auf, k Geraden durch je zwei Punkte aus P zu repräsentieren und prüft, ob diese Geraden alle Punkte treffen. Dies geschieht mit Hilfe eines beschränkten Suchbaumes wie im Folgenden beschrieben.

Wir betrachten die Punkte P in einer festen Reihenfolge $P = \langle p_1, p_2, \ldots, p_n \rangle$ und beginnen, wie im Hinweis angegeben, bei einem leeren Knoten mit k Einträgen. Da jeder Knoten durch eine der Geraden – definiert durch jeden der gefüllten k Einträge – überdeckt werden muss, stellt die Wahl der Reihenfolge keine Einschränkung da. Für einen beliebigen Knoten w_i der Suchbaumtiefe i können wir bis zu k Nachfolger der Suchbaumtiefe $i + 1$ generieren. Der j-te Nachfolger von w_i ($w_{i+1,j}$) wird durch eine Kopie von w_i erzeugt, bei der zusätzlich der nächste zu betrachtende Punkt in den Eintrag an Stelle j eingefügt wird. Besteht der Eintrag an Stelle j schon aus zwei Punkten, so wird $w_{i+1,j}$ verworfen und w_i erhält einen Nachfolger weniger. Wird der Eintrag an Stelle j komplett gefüllt, so wird durch die beiden Punkte eine neue Gerade definiert und alle von ihr überdeckten Knoten aus der Liste der zu bearbeitenden Punkte entfernt (für den aktuellen Teilbaum). Auf diese Weise werden alle Möglichkeiten getestet, k verschiedene Geraden aus den Punkten zu erzeugen. Jeder Blattknoten auf der Suchbaumtiefe $2k$ definiert nun k verschiedene Geraden. Sind bei einem dieser Knoten alle Punkte überdeckt, so kann eine Erfolgsmeldung ausgegeben werden.

d) Der Verzweigungsgrad des Suchbaums ist höchstens k, da es in jedem Schritt nur k mögliche Einträge zu füllen gibt. Die Suchbbaumtiefe ist höchstens $2k$, da nach Auswahl von $2k$ Knoten alle Einträge gefüllt sind. Damit hat der Suchbaum $O(k^{2k})$ Knoten. Wird kein Eintrag voll, so werden beim Erzeugen eines Knotens Kosten von $O(1)$ erzeugt. Ist dagegen eine Überprüfung der verbleibenden Knoten nötig, so entstehen Kosten von $O(n)$ für das Überprüfen der maximal $O(n)$ verbleibenden Punkte. Damit ist $O(\frac{k^{2k}}{2} \cdot n)$ eine obere Schranke für die Gesamtlaufzeit. Daraus folgt, dass das *line shooting* Problem *fixed parameter tractable* ist.

Aufgabe 5 (*Analyse+Entwurf: Buffetplanung*)

Sie sind beauftragt worden, das Buffet für eine große Gala zu organisieren. Damit jeder geladene Gast auch eine seiner Lieblingsspeisen vorfindet, durften alle n Gäste eine Auswahl an d Gerichten angeben, die ihnen besonders schmecken. Der bisher für den Aufbau des Buffets vorgesehene Platz bietet allerdings nur Platz für k verschiedene Gerichte. Sie müssen nun entscheiden, ob der Platz ausreicht, so dass jeder Gast mindestens eines seiner angegebenen Gerichte vorfindet.

a) Formulieren Sie die Aufgabe als graphentheoretisches Problem (Stichwort: *Hypergraphen*).

b) Entwerfen Sie einen Algorithmus, der Ihr Entscheidungsproblem löst.

c) Stellen Sie eine Rekurrenzrelation für die Laufzeit Ihres Algorithmus auf und lösen Sie diese.

d) Ist Ihr Algorithmus *fixed parameter tractable*? Begründen Sie kurz.

Musterlösung:

a) Es handelt sich um ein *hitting set* Problem. Sei G die Menge an Gerichten und P die Menge an Personen mit $p \in P \Leftrightarrow p \subseteq G, |p| \leq d$. Gesucht ist nach einer Teilmenge $H \subseteq G, |H| \leq k$ mit $\forall p \in P : \exists g \in H : g \in p$. Jede Person wird also über ihre angegebenen Gerichte charakterisiert und gesucht ist eine Teilmenge aller Gerichte, die mindestens ein angegebenes Gericht jeder Person enthält.

   Dies entspricht einem *vertex cover* Problem auf Hypergraphen. Ein Hypergraph $H(V, E)$ ist ein verallgemeinerter Graph, bei dem jede Kante mehr als zwei Knoten enthalten kann. Hier entsprechen Kanten den Personen in der Aufgabenstellung und Knoten den einzelnen Gerichten.

b) Der Algorithmus funktioniert analog zu dem Algorithmus, der entscheidet, ob für einen Graphen ein *vertex cover* der Größe k existiert.

```
function hasHittingSet(H = (V, E) : Hypergraph, k : Integer)
    if |V| < k then return false
    else if E = ∅ then return true
    else if k = 0 then return 0
    else
        choose e ∈ E
        for all v ∈ e do
            H_v = (V \ {v}, {e ∈ E | v ∉ e})
            if hasHittingSet(H_v, k − 1) then return true
            end if
        end for
    end if
end function
```

c) Die Rekurrenzrelation ergibt sich zu

$$\begin{aligned} T(0, n, d) &= O(1), \\ T(k, n, d) &= d \cdot T(k-1, n, d) + O(n). \end{aligned}$$

   In jedem Schritt werden maximal d rekursive Aufrufe durchgeführt und man benötigt $O(n)$ Zeit, um den Hypergraph zu verkleinern. Der Basisfall benötigt konstante Zeit. Löst man die Rekurrenz auf, ergibt sich die Abschätzung

$$T(k, n, d) = O(d^k \cdot n).$$

d) Ja, der Algorithmus ist *fixed parameter tractable*, da die Laufzeit in $O(d^k \cdot n)$ liegt und somit polynomiell in n und davon unabhängig abhängig von k ist.

Aufgabe 6 (*Analyse: ADAC Mitgliedschaft*)

Der "**A**utomobil **D**urch **A**lgorithmiker **C**lub" (ADAC) leistet auf Autobahnen Pannenhilfe. Ein Autofahrer hat in seiner Zeit als Verkehrsteilnehmer n Pannen, $n \in \mathbb{N}_{\geq 0}$, für die er die Hilfe des ADAC in Anspruch nehmen muss. Für jede geleistete Pannenhilfe verlangt der Club eine Aufwandsentschädigung abhängig von der Schwere der Panne. Mitglieder beim ADAC müssen lediglich ein Viertel dieser Kosten bezahlen. Eine lebenslange Mitgliedschaft kann man sich durch eine Einmalzahlung in Höhe von 1000 DM (**D**ijkstra **M**ark) sichern.
Da nicht schon mit Erwerb des Führerscheins klar ist, wie viele Pannen man in seinem Leben haben wird und wie schwerwiegend diese sein werden, stellt sich die Frage, ab wann es sich lohnt eine Mitgliedschaft beim ADAC zu erwerben.

a) Geben Sie eine Strategie an, die einen kompetitiven Faktor (competitive ratio) von ∞ erreicht. Begründen Sie kurz.

b) Wie gut ist die Strategie, sich nie eine Mitgliedschaft beim ADAC zu sichern? Begründen Sie.

c) Zeigen Sie, dass folgende Strategie einen kompetitiven Faktor $c = 3$ hat. Die Strategie ist, sich beim Pannenhelfer eine Mitgliedschaft zu kaufen, wenn die momentan von ihm bearbeitete Panne die Gesamtausgaben für Pannen (ohne Mitgliedschaft) auf über 500 DM anheben würde.

Hinweis: Verwenden Sie die summierten Gesamtkosten K über alle Pannen (ohne Mitgliederrabatt).

Musterlösung:
Ein Algorithmus ALG wird als *streng c-kompetitiv* bezeichnet, wenn für alle Eingaben I gilt

$$c = \sup_I \frac{ALG(I)}{OPT(I)}$$

Dieser Wert wird als *kompetitiver Faktor* (*competitive ratio*) bezeichnet.

Der Übersichtlichkeit halber wird im Folgenden ohne Einheiten gerechnet:

a) Wenn man sofort mit Erhalt der Fahrerlaubnis eine Mitgliedschaft beim ADAC erwirbt, gibt man im schlimmsten Fall 1000 DM aus, nimmt aber die Hilfe des ADAC nie in Anspruch. Dies ergibt einen kompetitiven Faktor von $c = \frac{1000}{0} = \infty$.

b) Wenn man sich nie eine Mitgliedschaft kauft, gibt man offensichtlich höchstens 4 mal soviel für den ADAC aus wie ein Mitglied. Die summierten Gesamtkosten für alle Pannen seien mit K bezeichnet. Für $K \to \infty$ konvergiert der kompetitive Faktor gegen $c = \frac{K}{1000+K/4} \to 4$.

c) Die summierten Gesamtkosten für alle Pannen seien mit K und die summierten Kosten vor Beitritt zum ADAC mit $K_1 \leq 500$ bezeichnet. Die eigene Strategie liefert

$$ALG = \begin{cases} K_1 + 1000 + (K - K_1)/4 & K > 500, \\ K & \text{sonst} \end{cases}$$

in Abhängigkeit davon, ob man jemals über 500 DM Kosten für Pannen hat oder nicht. Die optimale Strategie ist durch

$$OPT = \begin{cases} 1000 + K/4 & K \geq 1333\frac{1}{3}, \\ K & \text{sonst} \end{cases}$$

gegeben. Entweder kauft man sich sofort eine Mitgliedschaft oder nie. Die Grenzkosten ergeben sich durch Lösen von $1000 + K/4 \stackrel{!}{=} K$. Der kompetitive Fakor ist der maximale Quotient von ALG und OPT. Allgemein gilt

$$\frac{ALG(K)}{OPT(K)} = \begin{cases} \frac{K}{K} & K \leq 500, \\ \frac{1375+K/4}{1000+K/4} & K \geq 1333\frac{1}{3}, \\ \frac{1375+K/4}{K} & \text{sonst} \end{cases}$$

(mit $K_1 = 500$ gesetzt, da wir nur am maximalen Wert interessiert sind). Das Supremum dieses Quotienten ist 3 für $K \to 500, K > 500$. Damit ist der kompetitive Faktor dieser Strategie

$$c = \sup_K \frac{ALG(K)}{OPT(K)} = 3.$$

Aufgabe 7 (*Analyse: Online-gaming-Algorithmen*)

Angestellte in einem Rechenzentrum haben einen recht eintönigen Job. Ihnen stehen zwar die größten Rechner zur Verfügung, Sie dürfen diese aber nicht selbst verwenden. Stattdessen heisst es nur, die Maschinen möglichst gut auszulasten und Rechnungen zu schreiben. Ein ziemlich langweiliger Job möchte man meinen – zum Glück gibt es noch Computerspiele.
Ein Mitarbeiter hat zu Weihnachten ein neues Computerspiel erhalten, dass er liebsten andauernd spielen würde. Diesem Wunsch steht leider seine Arbeit im Wege. Eine neue Dienstanweisung besagt, dass die teuren Großrechner zu mindestens 50% ausgelastet sein müssen. Da das Scheduling in diesem Rechenzentren noch von Hand durchgeführt wird, muss der spielfreudige Mitarbeiter die laufenden Jobs überwachen und schnell neue Jobs auf leerlaufende Maschinen verteilen. So bleibt leider nur wenig Zeit zum Spielen am Arbeitsplatz.
Jeder Job hat eine Mindestlaufzeit von 5 Minuten. Die Zeit zum Verteilen der Jobs kann für die Bestimmung der Auslastung vernachlässigt werden. Der Mitarbeiter kann in dieser Zeit aber nicht spielen. Ein Job hat während seiner Ausführung die Maschine exklusiv. Es gibt keine automatischen Benachrichtigungen über das Ende eines Jobs. Der Mitarbeiter muss dies selbst periodisch überprüfen.

a) Entwerfen Sie einen *online scheduling* Algorithmus, der die freie Zeit des Mitarbeiters unter Einhaltung der Nebenbedingungen maximiert, wenn er einen Großrechner zu betreuen hat.

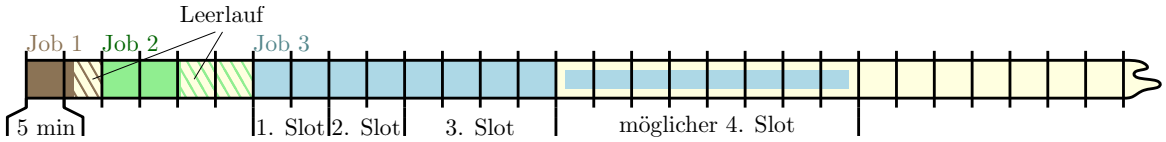
b) Zeigen Sie, dass Ihr Algorithmus optimal bzgl. der Freizeit des Mitarbeiters ist.

Musterlösung:

a) Wir definieren den Algorithmus induktiv. Durch die Mindestlaufzeit von 5 Minuten kann man jedem Job zunächst ein Zeitslot von 10 Minuten zuweisen. Nach Ablauf dieses Zeitslots muss der Mitarbeiter nachschauen, ob der Job bereits abgeschlossen ist, in der Zwischenzeit kann er spielen. Falls der Job in dieser Zeit beendet wurde, stand die Maschine höchstens 50% der Zeit still bis der Mitarbeiter dies erkannt und einen neuen Job gestartet hat. Andernfalls muss ein neuer Zeitslot für den noch laufenden Job zugewiesen werden.

Die Länge des neuen Zeitslots wird gleich der bisherigen Gesamtlaufzeit des Jobs (10 Minuten) gewählt. Dies verdoppelt die mögliche Gesamtlaufzeit des Jobs bis zur nächsten Überprüfung durch den Mitarbeiter auf 20 Minuten. Dadurch wird sichergestellt, dass der Rechner maximal die Hälfte der Zeit leerläuft. Wenn der Job ε Zeiteinheiten nach Start des neuen Zeitslots endet, ist er insgesamt $10+\varepsilon$ Minuten gelaufen und der Rechner damit zu $\frac{10+\varepsilon}{20} > 50\%$ ausgelastet gewesen. Sollte der neue Zeitslot auch nicht genügen, wird dieses Vorgehen wiederholt, bis der Job endet (neuer Zeitslot von $20, 40, \ldots$ Minuten, maximale Gesamtlaufzeit von $40, 80, \ldots$ Minuten).

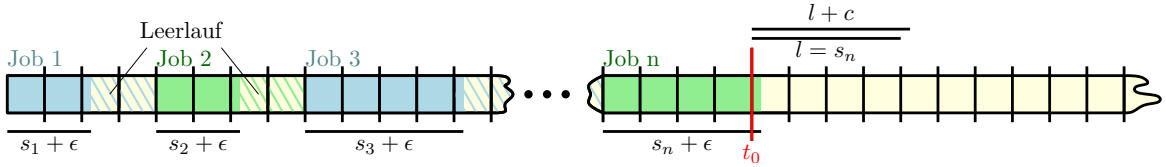
Folgende Abbildung verdeutlicht den Ablauf des Algorithmus:



Kein Job erhält mehr als das Doppelte seiner Laufzeit an Zeitslots zugeteilt. Somit ist der Großrechner im Durchschnitt zu mindestens 50% ausgelastet – wie gefordert.

Auch wenn die Aufgabenstellung einen eher scherzhaften Hintergund hat, so ist die vorgestellte Technik der Verdopplung (*doubling*) ein nützliches Hilfsmittel beim Entwurf von *online* Algorithmen. Insbesondere Algorithmen die Suchschritte benötigen, können durch derartige Techniken oft schon recht gute Approximationsgarantien (bzw. kompetitive Faktoren) liefern.

b) Angenommen, es existiere ein Algorithmus, der längere Zeitslots zwischen zwei Überprüfungen erlaubt als unser Algorithmus. Dann existiert für eine Folge an Jobs ein Zeitpunkt t_0, an dem die zugeteilten Zeitslots beider Algorithmen erstmals voneinander abweichen. Unser Algorithmus weise einen Zeitslot der Länge l zu, der potentiell bessere Algorithmus einen Zeitslot der Länge $l' = l + c$. Betrachte konkret eine Folge an Jobs, für die unser Algorithmus genau die minimale Auslastungsgrenze von 50% einhält (siehe Grafik).



Job i benötigt Zeit $s_i + \epsilon$ mit s_i gleich einer Dauer, nach der eine Überprüfung stattfindet – also nach $10, 20, 40, 80, \ldots$ Minuten. Zum Zeitpunkt t_0 weist unser Algorithmus einen Zeitslot von $l = s_n$ als Verlängerung zu, der andere Algorithmus einen Zeitslot von $l' = l + c = s_n + c$. Wähle $\epsilon < \frac{c}{2n}$, so ergibt sich nach Abarbeitung dieses Zeitslots eine Auslastung von

$$\frac{\sum_{i=1}^{n} s_i + \epsilon}{c + \sum_{i=1}^{n} 2s_i} = \frac{n \cdot \epsilon + \sum_{i=1}^{n} s_i}{c + 2 \cdot \sum_{i=1}^{n} s_i} < \frac{\frac{c}{2} + \sum_{i=1}^{n} s_i}{2 \cdot \left(\frac{c}{2} + \cdot \sum_{i=1}^{n} s_i\right)} = \frac{1}{2}$$

für den alternativen Algorithmus. Beachte: Startet zu dem Zeitpunkt t_0 ein neuer Job, so ist das Ergebnis analog zu erreichen durch eine Wahl von $s_n = 10$, der doppelten Mindestlaufzeit des Algorithmus. Damit hält er nicht – wie gefordert – eine minimale Auslastung von 50% ein. Unser Algorithmus verwendet also bereits die maximal möglichen Zeitslots zwischen zwei Überprüfungen und ermöglicht dem Mitarbeiter die meiste Freizeit.