In diesem Video betrachten wir nicht ausschließlich, aber überwiegend algorithmische Probleme, die in gewisser Weise schwer sind. Wir setzen das Video zur Komplexität algorithmischer Probleme daher voraus, aber im Wesentlichen lässt sich das vorliegende Video auch aus sich selbst heraus verstehen.

Im Video zur Komplexität algorithmischer Problemstellungen haben wir aus theoretischer Sicht betrachtet, was es konkret heißt, dass Probleme schwer sind. Hier noch einmal die praktischen Konsequenzen.

Das ist die Misere aus praktischer Sicht. Leider trifft diese Aussage auf die meisten algorithmischen Probleme aus der Praxis zu.

Und das ist der Aspekt, den wir in diesem Video jetzt angehen werden: dass wir trotzdem irgendwie doch zu vernünftigen Algorithmen kommen. Aus Zeitgründen betrachten wir nur ein paar einfache Konzepte und die auch nur sehr oberflächlich. In weiterführenden Lehrveranstaltungen haben Sie die Gelegenheit, tiefer in dieses wichtige Thema einzutauchen.

======

Als erstes allgemeines Konzept betrachten wir mehrstufige Herangehensweise, was im Einzelfall drei- oder vierstufig oder noch mehr Stufen heißen kann. Aber der zweistufige Fall ist wohl am häufigsten, und zur Erläuterung des Grundgedankens reichen auch Fälle mit zwei Stufen.

Die erste Stufe ist eine Art Vorstufe, ein *Preprocessing*. Wir schauen uns drei wichtige Strategien auf den nächsten Folien anhand konkreter Beispiele an. Letztendlich geht es in der ersten Stufe immer um dasselbe: Reduktion der Komplexität, Komplexität sowohl im landläufigen Sinne wie auch in unserem spezifischen Sinn asymptotische Komplexität.

Reduktion der schieren Datenmenge ist eine einfache und natürlich auch eine nützliche Form von Reduktion der Komplexität.

Nicht alle Details in einem Problem sind gleichermaßen kritisch für die Gesamtlösung. Es ist oft sinnvoll, die kritischen Details vorab zu behandeln und danach dann den unkritischen Rest.

Last not least lässt sich die Komplexität auch dadurch reduzieren, dass die Details der Lösung zuerst nur ungefähr festgelegt werden, erst im zweiten Schritt dann exakt.

In jedem Fall wird das Ergebnis aus der ersten Stufe als Basis genommen, um in der zweiten Stufe dann das algorithmische Problem vollständig zu lösen.

======

Ein konkretes, außergewöhnlich effektives Beispiel aus meiner eigenen Praxis, bearbeitet Mitte der neunziger Jahre in Kooperation mit der Deutschen Bahn. Gegeben ist eine Menge von Zugläufen, wobei hier an jedem Zuglauf eigentlich nur die Menge der Haltebahnhöfe überhaupt interessiert. Gesucht ist eine möglichst kleine Menge von Bahnhöfen, so dass jeder Zug an mindestens einem dieser Bahnhöfe hält.

Links sehen Sie eine optimale Lösung für die Gesamtmenge aller ICE-Zugläufe seinerzeit, unschwer zu erkennen als Berlin, Hannover, Köln, Frankfurt, Stuttgart und München. Der Punkt ist: Allein die Datenreduktion, die wir gleich betrachten werden, hat die Gesamtheit aller ICE-Zugläufe reduziert auf diese sechs Bahnhöfe. Es sind sechs Zugläufe übrig geblieben, jeder wurde auf einen dieser Bahnhöfe reduziert.

Rechts sehen Sie das Ergebnis der Datenreduktion für *alle* Zugläufe der Deutschen Bahn seinerzeit. An den allermeisten Stellen wurden die Daten so wie links tatsächlich auf einzelne isolierte Bahnhöfe reduziert, aber nicht an allen Stellen. An manchen Stellen hat die Datenreduktion zu kleinen Zusammenhangskomponenten geführt, die aus mehreren Bahnhöfen bestehen, zusammengehalten von Zügen, die nicht auf einen, sondern nur auf zwei oder drei Bahnhöfe reduziert werden konnten. Die isolierten Bahnhöfe plus je eine optimale Auswahl in jeder Zusammenhangskomponente ergeben eine optimale Lösung.

======

Die Regeln zur Datenreduktion, mit denen dieser außergewöhnlich drastische Effekt erreicht werden konnte, sind überraschend simpel und eigentlich recht naheliegend.

Nehmen wir etwa die Situation Darmstadt Hauptbahnhof und Darmstadt Süd. Alle Züge, die in Darmstadt Süd halten, halten auch in Darmstadt Hauptbahnhof.

Also können wir Darmstadt Süd aus der Datenbasis herausnehmen. Denn sollte es eine optimale Lösung geben, in der Darmstadt Süd enthalten ist, dann könnten wir darin einfach Darmstadt Süd durch Darmstadt Hauptbahnhof ersetzen und würden eine gleich große, also ebenfalls optimale Lösung erhalten, die ebenfalls alle Zugläufe überdeckt.

Analog bei Zügen statt Bahnhöfen. Betrachten Sie beispielsweise zwei Züge, die auf derselben Strecke fahren, aber der eine Zug hält an jedem Bahnhof, der andere Zug fährt durch einige Bahnhöfe ohne Halt hindurch.

Dann kann der erste Zug aus der Datenbasis herausgenommen werden, denn jeder Bahnhof, der den zweiten Zug überdeckt, überdeckt automatisch auch den ersten.

Nehmen wir also zuerst möglichst viele Bahnhöfe mit der ersten Regel heraus, danach möglichst viele Züge mit der zweiten Regel. Danach wird es meist neue Möglichkeiten geben, die erste Regel anzuwenden, nach Anwendung der ersten Regel wird es wieder neue Möglichkeiten für die zweite Regel geben, und so weiter. Diese beiden Regeln werden daher immer abwechselnd angewandt, solange bis keine der beiden Regeln mehr anwendbar ist. Das Endergebnis haben Sie gesehen.

======

Jetzt die zweite erwähnte Möglichkeit, die Komplexität vorab zu reduzieren.

In Großbetrieben ist es recht typisch, dass einige Ressourcen sogenannte *Engpassressourcen* sind.

Zum Beispiel Maschinen. Eine Engpassmaschine ist eine, die so viele Aufträge zu bearbeiten hat, dass nicht viel Spielraum bei der zeitlichen Einplanung bleibt. Die Einplanung auf dieser Maschine muss also besonders gut überlegt sein.

Die Idee ist, zuerst die Abarbeitung der Aufträge auf den Engpassmaschinen zu planen und erst danach die Arbeitspläne der anderen Maschinen zu erstellen. Denn die anderen Maschinen, die eben keine Engpassmaschinen sind, haben ja per Definition größeren Spielraum, so dass die Planung auf den Engpassmaschinen problemlos auf die anderen Maschinen erweiterbar sein dürfte.

Oft ist es auch der Fall, dass eine Maschine nur zu bestimmten, einigermaßen gut vorhersehbaren Zeiten zum Engpass wird. Dann würde man diese Maschine nur in diesen Zeiten in der ersten Stufe berücksichtigen.

Anderes Beispiel: In einigen Großbetrieben gibt es eine Betriebseisenbahn, manchmal sogar sehr weit verzweigt.

Auch hier sind Engpassressourcen typisch, also Strecken und Knotenpunkte, wo absehbar so viel Verkehr drüber laufen muss, dass kaum Spielraum für die zeitliche Einplanung bleibt.

======

Ein ganz anderes Planungsproblem, aber wieder dieselbe zweistufige Grundidee:

Für das nächste Semester soll jeder wöchentlichen oder zweiwöchentlichen Lehrveranstaltung ein Hörsaal, ein Wochentag und eine Uhrzeit zugewiesen werden. Falls eine Lehrveranstaltung mehrere Termine hat, entsprechend viele Tripel, jeweils bestehend aus Hörsaal, Tag und Uhrzeit.

Analog zum Beispiel Großbetrieb gibt es auch hier kritische Punkte, nämlich Lehrveranstaltungen, an denen viel dranhängt.

Vor allem große Lehrveranstaltungen, die Pflicht in mehreren Studiengängen sind und daher zu vielen anderen Lehrveranstaltungen nicht zeitgleich geplant werden dürfen. Alle diese Lehrveranstaltungen mit ihren gegenseitigen Ausschließungen ergeben ein Geflecht, das am Besten vorab als Ganzes in der ersten Stufe eingeplant wird.

Alle anderen Lehrveranstaltungen sind dann viel flexibler und wahrscheinlich relativ problemlos einplanbar. Das kann dann in einer zweiten Stufe nachgeholt werden.

======

Noch ein anderes Beispiel kommt aus der Technischen Informatik. Auf einem Mikrochip soll eine große Zahl von Verbindungen verdrahtet werden, so dass sie sich gegenseitig nicht berühren. Natürlich sind noch diverse weitere Bedingungen einzuhalten. Die Zeit, bis nach einem Signal wieder ein ausreichend eingeschwungener Zustand erreicht ist, ist zu minimieren.

Sehr grob gesprochen, sind die längsten Verbindungen die hierfür kritischen. Man weiß nicht vorher, welche das sein werden. Aber man kommt im allgemeinen zu guten Lösungen, wenn man zuerst diejenigen Verbindungen festlegt, bei denen die Pins weit auseinander liegen. Die verdrahtet man in der ersten Stufe so kurz wie möglich, und erst danach, in der zweiten Stufe, zieht man alle anderen Verbindungen drum herum.

======

Eine ganz andere Form der Vorabverarbeitung, die dritte Idee auf unserer Eingangsliste, kann man am selben Anwendungsbeispiel Verdrahtung von Mikrochips ebenfalls gut illustrieren.

======

Die vier Punkte sind im Beispiel miteinander zu verbinden. Die Mikrochipfläche wird in rechteckige Zonen zerlegt, und für alle weiträumig zu verlegenden Verbindungen wird in der ersten Stufe zunächst einmal nur festgelegt, durch welche Zonen sie laufen sollen, also grob gesprochen, ob sie an den einzelnen funktionellen Komponenten auf dem Chip jeweils rechts oder links vorbei gelegt werden sollen. In der zweiten Stufe werden dann alle Verbindungen – die weiträumigen und die lokalen – Zone für im Detail verlegt.

======

Auch in anderen Kontexten macht es Sinn, die Lösung in einer ersten Stufe nur grob und dann in einer zweiten Stufedetailliert zu berechnen.

Wieder Großbetriebe als Beispiel, konkret Aufgabenplanung in einem Zeithorizont.

Jeder Aufgabe wird in der ersten Stufe erst einmal nur ein großzügig bemessenes Zeitintervall zugeordnet, in dem sie irgendwann erledigt wird, und erst in der zweiten Stufe werden alle Aufgaben exakt eingeplant. Oder alternativ doch schon einen exakten Zeitpunkt in der ersten Stufe, aber der darf in der zweiten Stufe dann noch nachkorrigiert werden, falls nötig.

Die Arbeitslast sollte in der ersten Stufe so auf den Zeithorizont verteilt werden, dass überall noch Zeitpuffer bleiben, die einerseits für die Detailplanung in der zweiten Stufe notwendig werden könnten. Andererseits sind Zeitpuffer natürlich wichtig zum Abfedern unvorhergesehener Ereignisse. Und da man naturgemäß nicht weiß, wann ein unvorhergesehenes Ereignis eintritt, verteilt man die Puffer am Besten gleichmäßig.

Außer man kann vorab Zeitintervalle identifizieren, in denen mit höherer Wahrscheinlichkeit als sonst ein Problem auftreten wird. In einem solchen Zeitintervall sollten dann natürlich besonders großzügig Puffer eingeplant werden.

======

Damit haben wir Beispiele für alle drei hier betrachteten Ideen für zweistufige Ansätze gesehen. Wir schließen dieses Thema ab und wenden uns einem ganz anders gearteten algorithmischen Konzept zu, dass Sie schon kennen gelernt haben, ohne dass wir ihm einen Namen gegeben haben.

In vielen algorithmischen Problemstellungen lässt sich eine Eingabe sinnvoll in zwei oder mehr Teile zerlegen, die rekursiv auf dieselbe Art behandelbar sind. Das macht immer dann Sinn, wenn die Teilergebnisse effizient zu einem Gesamtergebnis zusammengefügt werden können.

Die Teilergebnisse lassen sich wiederum rekursiv durch weiteres Aufteilen der Eingabe und Zusammenfügen von deren Teilergebnissen berechnen und so weiter, bis die Teile so klein werden, dass das Ergebnis trivial oder zumindest ohne weitere Aufteilung berechnet werden kann.

Mergesort und Quicksort sind zwei Beispiele. Die Eingabesequenz wird rekursiv solange aufgeteilt, bis die Einzelteile maximal ein Element enthalten. Zusammengefügt werden die Teilsequenzen bei Mergesort dann durch die Subroutine Merge, bei Quicksort einfach durch Konkatenation.

Für viele algorithmische Problemstellungen im *geometrischen* Bereich kann man Algorithmen nach dem Prinzip Divide-and-Conquer entwickeln, wir schauen uns ein elementares Beispiel auf der nächsten Folie an.

======

Wir betrachten hier nur den zweidimensionalen Fall. Einen entsprechenden, aber wesentlich komplizierteren Algorithmus gibt es auch in 3D. Es gibt genau ein flächenminimales Polygon, das endlich viele Punkte in der Ebene umspannt, in 3D ein volumenminimales Polytop. Dies nennt man in beiden Fällen die *konvexe Hülle* der Punkte.

Die Punktmenge wird an einer der Koordinatenachsen in zwei ungefähr gleich große Teilmengen zerlegt. Für die beiden Teilmengen wird jeweils die konvexe Hülle berechnet, und dann werden die konvexen Hüllen der beiden Teilmengen zur konvexen Hülle der Gesamtpunktmenge zusammengefügt. Als Rekursionsbasis in 2D kann man gut ein-, zwei oder auch dreielementige Punktmengen nehmen.

Wegen der Aufteilung ungefähr halbe-halbe ist die Rekursionstiefe natürlich wieder einmal logarithmisch. Die asymptotische Komplexität ist daher n-log-n.

======

Noch ein ganz anderes Konzept, das wir in dieser Lehrveranstaltung bisher noch nicht gesehen haben. Der Begriff „dynamische Programmierung“ ist älter als Computer und Informatik und hat nichts mit dem Programmieren von Computern zu tun.

In vielen rekursiven Algorithmen wird die asymptotische Komplexität und damit auch das reale Laufzeitverhalten exponentiell aufgebläht dadurch, dass dieselben Teilergebnisse in verschiedenen Knoten des Rekursionsbaumes immer und immer wieder neu berechnet und dann wieder vergessen werden.

Dynamische Programmierung vermeidet das, indem nicht rekursiv, sondern iterativ und bottom-up alle potentiell relevanten Teilergebnisse Schritt für Schritt berechnet werden, von klein nach groß, so dass jedes Teilergebnis abrufbereit zur Verfügung steht, wenn es gebraucht wird.

Wir schauen uns hier nur zwei sehr einfache Beispiele an, die Sie vielleicht auch schon kennen. Später, unter dem Stichwort Granularität, schauen wir uns noch ein weiteres, komplizierteres Beispiel an.

======

Hier also die beiden Beispiele.

Das Beispiel Fibonacci-Zahlen ist so einfach, dass es meist gar nicht als Beispiel für dynamische Programmierung wahrgenommen wird.

Binomialkoeffizienten hingegen sind ein Standardbeispiel zur Einführung in die dynamische Programmierung.

======

Dies ist die iterative Umsetzung der Rekursionsgleichung für Fibonacci-Zahlen.

Jedes Zwischenergebnis wird nur in den nächsten beiden Iterationen benötigt, nachdem es berechnet wurde, und kann danach überschrieben werden.

======

Die iterative Berechnung von Binomialkoeffizienten schauen wir uns in zwei Varianten an. Auf dieser Folie sehen wir die einfachere Variante, in der alle Zwischenergebnisse in einer Matrix abgespeichert sind und während des gesamten Algorithmus zur Verfügung stehen, sobald sie einmal berechnet worden sind. Wir betrachten nur den Normalfall, also 0 <= k <= n.

Schauen Sie sich die relevanten Werte selbst im Pascalschen Dreieck an, um diese iterative Berechnung nachzuvollziehen.

======

Auch bei Binomialkoeffizienten wird jedes Zwischenergebnis nur kurzzeitig benötigt. Zudem haben wir in der Variante von eben etliche Zwischenergebnisse berechnet, die wir für das Endergebnis überhaupt nicht benötigen. Das geht auch besser.

Dank der Symmetrie des Binomialkoeffizienten ändert diese Zeile nichts am Ergebnis. Sie können Ihr Verständnis der folgenden Implementation überprüfen anhand der Frage, warum diese Zeile da steht und nicht ersatzlos gestrichen werden kann.

Da jedes Zwischenergebnis nur kurzzeitig benötigt wird, reicht ein Array anstelle einer Matrix. Im Pascalschen Dreieck bilden die tatsächlich benötigten Zwischenergebnisse ein auf der Spitze stehendes Rechteck mit den Eckpunkten 0 über 0 und n über k sowie n minus k halbe abgerundet über 0 und k halbe abgerundet über k halbe abgerundet.

Das Array durchläuft dieses Rechteck von rechts oben nach links unten, am Anfang stehen also Werte drin von der Form i über i.

Machen Sie sich klar, dass die Reihenfolge essentiell ist, in der die Array-Komponenten überschrieben werden, immer von links oben nach rechts unten. Machen Sie sich weiter klar, dass die Berechnungsformel daher exakt die Rekursionsformel für Binomialkoeffizienten ist.

Wie gesagt, ist n über k die unterste Ecke des Rechtecks, am Ende also die Array-Komponente mit dem höchsten Index.

======

Ein völlig anderer, intuitiv sehr naheliegender Ansatz heißt „greedy“, also deutsch „gierig“. Meines Erachtens trifft es das Wort „kurzsichtig“ im Deutschen aber besser.

Wir betrachten zuerst die Standardform, später eine Verallgemeinerung. Die Standardform ist immer dann anwendbar, wenn es darum geht, eine möglichst gute Auswahl zu treffen. Zum Beispiel das Rucksackproblem ist offenkundig von dieser Art, das heißt, endlich viele Objekte stehen zur Auswahl, jedes Objekt hat ein Gewicht und einen Profitwert, und wir wollen eine Auswahl treffen, die das maximale Gesamtgewicht nicht überschreitet, unter dieser Einschränkung uns aber einen möglichst guten Gesamtprofit bringt.

Aber auch die anderen $\mathcal{NP}$-schweren Probleme aus dem Video zur Komplexität algorithmischer Probleme kann man so interpretieren. Zum Beispiel das TSP oder das Steinerbaumproblem: In beiden Problemen soll eine möglichst billige Auswahl von Kanten getroffen werden, so dass die Auswahl insgesamt eine Rundtour beziehungsweise einen Steinerbaum ergibt.

Oder auch die algorithmischen Probleme Mengenpackung und Mengenüberdeckung aus diesem Video: Hier suchen wir eine Auswahl von Teilmengen, die die Grundmenge überdecken. Und beim Problem Clique beziehungsweise unabhängige Menge suchen wir eine Auswahl von Knoten, die paarweise verbunden beziehungsweise paarweise unverbunden sind.

Wir sehen, sehr viele verschiedenartige Optimierungsprobleme kann man interpretieren als das Problem, eine optimale Auswahl unter gewissen problemspezifischen Nebenbedingungen zu treffen. Daher lässt sich der Greedy-Algorithmus schon in seiner eingeschränkten Standardform auf sehr viele verschiedene Optimierungsprobleme anwenden.

Der Greedy-Algorithmus ist auf natürliche Weise formulierbar als eine Schleife. Das Vorgehen ist im Grunde bei jedem Optimierungsproblem immer dasselbe.

Vor der allerersten Iteration ist noch kein Element ausgewählt.

In jeder Iteration geht der Algorithmus alle noch nicht betrachteten Objekte durch, und zwar konkret diejenigen Objekte, deren Hinzunahme immer noch eine zulässige Lösung erlaubt. Aus diesen wählt er das in diesem Moment profitabelste.

Die Invariante ist also, dass – vor und nach jeder Iteration – die momentane Auswahl eine zulässige Lösung ist oder zu einer zulässigen Lösung erweiterbar ist.

======

Schauen wir uns anhand von ein paar Beispielen aus dem Video über die Komplexität algorithmischer Probleme an, was das konkret bedeutet.

Beim Rucksackproblem war der Profit eines Objektes schon expliziter Teil der Aufgabenstellung. Die Invariante ist hier, dass das maximale Gesamtgewicht nicht überschritten ist. In jeder Iteration fügen wir also das profitabelste Objekt in die Auswahl ein, das zusammen mit der bisherigen Auswahl nicht das maximale Gesamtgewicht überschreitet.

Beim TSP ist die Länge einer Kante sozusagen der negative Profit, wir präferieren also Kanten mit geringen Längen. Invariante ist, dass keine zwei Kanten aus demselben Knoten herauszeigen, keine zwei Kanten in denselben Knoten hineinzeigen und kein Zykel auf einer echten Teilmenge der Punktmenge geschlossen wird. Die leere Auswahl vor der ersten Iteration erfüllt die Invariante trivialerweise. Wird die Invariante durch entsprechende Auswahl der Kante in jeder Iteration durchgehalten, kommt am Ende zwangsläufig eine zulässige Lösung, eine Rundtour heraus.

Das TSP ist ein Beispiel dafür, dass man den Greedy-Ansatz auf derselben algorithmischen Problemstellung ganz unterschiedlich realisieren kann. Invariante und Variante erklären wie immer die algorithmische Idee: Die *Invariante* ist, dass die Kantenauswahl vor und nach jeder Iteration einen gerichteten Pfad auf den Punkten bildet, der zykelfrei ist, solange er noch nicht alle Punkte enthält. Daraus ergibt sich wieder zwangsläufig, dass das Endergebnis eine zulässige Lösung, eine Rundtour ist. Die *Variante* ist, dass am Endpunkt des Pfades ein neuer Punkt vorne angehängt wird. Dafür ist ein Punkt auszuwählen, der noch nicht im Pfad ist. Natürlich wählen wir gemäß Greedy-Idee immer den Punkt, zu dem die Entfernung vom bisherigen Endpunkt des Pfades am geringsten ist.

Beim Steinerbaumproblem wird es etwas unübersichtlich, wenn wir in jeder Iteration immer nur eine einzelne Kante hinzunehmen. Es macht mehr Sinn, immer gleich ganze Pfade einzufügen. Invariante ist dann, dass vor und nach jeder Iteration die ausgewählten Kanten einen Steinerbaum auf einer Teilmenge der Terminalmenge bilden. Die Variante ist, dass diese Teilmenge der Terminalmenge immer um eins größer wird. In jeder Iteration suchen wir also einen kürzesten Pfad von einem noch nicht ausgewählten Terminal zu irgendeinem Knoten – Terminal oder nicht – auf dem bisher konstruierten Steinerbaum.

Wir betrachten noch einmal kurz das Minimum Spanning Tree Problem, das MST, das ja der Spezialfall des Steinerbaumproblems ist, bei dem alle Knoten des Graphen Terminale sind. Der soeben beschriebene Greedy-Ansatz reduziert sich beim MST gerade zum Algorithmus von Prim. Aber auch der Algorithmus von Kruskal ist ein Greedy-Ansatz, nämlich genau dieses Auswählen von einzelnen Kanten, das wir soeben für das Steinerbaumproblem als inpraktikabel verworfen haben. Beim MST ist dieses Vorgehen hingegen sehr gut praktikabel.

Bei Mengenpackung und Mengenüberdeckung macht es Sinn, als Profit einer Teilmenge die Anzahl der Elemente der Grundmenge herzunehmen, die zwar in dieser Teilmenge, aber nicht in den vorher schon ausgewählten Teilmengen enthalten sind. Dies ist ein Beispiel dafür, dass der Profit einer Auswahlmöglichkeit nicht unbedingt von Anfang an ein für allemal feststeht, sondern sich erst während des Algorithmus aus den bisherigen Auswahlen ergibt und sich dann auch von Iteration zu Iteration ändern kann.

Bei Cliquen und unabhängigen Mengen kommen in jedem Auswahlschritt natürlich nur Knoten infrage, die mit allen bisher ausgewählten Knoten verbunden beziehungsweise mit allen bisher ausgewählten Knoten *nicht* verbunden sind. Hier gibt es wohl keine naheliegende Definition des Profits. Man könnte etwa als Profit eines Knotens die Anzahl von Knoten hernehmen, die sowohl mit diesem als auch mit den bisher ausgewählten Knoten verbunden beziehungsweise *nicht* verbunden sind. Das ist eine obere Schranke für die nach Hinzunahme des Knotens maximal noch mögliche Größe einer Clique beziehungsweise unabhängige Menge, stellvertretend für die wahre noch mögliche Größe.

======

Wir haben nun einerseits Beispiele gesehen, wo der Profit jedes Objekts ein für allemal feststeht, und andererseits Beispiele, wo der Profit eines Objekts sich von Iteration zu Iteration verändern kann.

In Fällen, in denen der Profit eines Objekts konstant bleibt, sich also nicht während der Schleife ändert, kann man die asymptotische Komplexität einer Schleifeniteration um einen linearen Faktor reduzieren.

Dazu werden einfach vorab alle Objekte absteigend nach Ihrem Profit sortiert. Das können wir, da der Profit jedes Objekts eben von Anfang an schon feststeht.

Der Punkt ist, wir sehen uns *nicht* in jeder Iteration alle noch verbliebenen Objekte an, sondern nur das jeweils nächste in dieser Sortierreihenfolge.

Wenn Hinzufügung dieses Objekts die Invariante nicht verletzt, dann fügen wir es hinzu. Ansonsten fügen wir es eben nicht hinzu und betrachten es natürlich auch in den weiteren Iterationen nicht mehr – einmal verworfen, für immer verworfen.

======

Greedy ist also ein sehr einfaches, naheliegendes Konzept.
Aber wie gut funktioniert es in der Praxis?

Die Erfahrung sagt, dass man mit Greedy häufig schon recht ordentliche Ergebnisse bekommen kann. Wie soeben anhand der verschiedenen Beispiele andiskutiert, hat man durchaus gewisse Freiheiten bei der Definition der Profite der einzelnen Elemente. Ich kann hier nicht ins Detail gehen, das würde zu weit führen, aber generell kann man diese Freiheiten dafür nutzen, die Ergebnisse des Greedy-Algorithmus zu verbessern.

Wir haben den Algorithmus von Kruskal für minimal spannende Bäume kennengelernt. Machen Sie sich klar, dass Kruskal tatsächlich ein Greedy-Algorithmus ist.

Das bedeutet, für das Problem, einen minimalen spannenden Baum zu finden, haben wir schon früher bewiesen, dass dieser doch eigentlich recht einfache Greedy-Ansatz immer eine optimale Lösung liefert.

Leider ist das eher die seltene Ausnahme, nicht die Regel. Aus praktischer Sicht ist der Greedy-Algorithmus häufig sehr gut, aber theoretisch-mathematisch kann für die allermeisten Fälle im Grunde nichts ausgesagt werden.

Was hat das MST und was haben ein paar weitere einzelne Problemstellungen, was die Mehrzahl aller Problemstellungen nicht hat? Der Unterschied ist, dass die ersteren die sogenannte *Matroidstruktur* aufweisen, die letzteren nicht. Was das genau ist und warum dadurch Optimalität der Greedy-Lösungen garantiert wird, das wird in weiterführenden Lehrveranstaltungen betrachtet.

======

Greedy war ja geeignet speziell für Problemstellungen, in denen die Lösungen Auswahlen aus einer Grundmenge sind. Die Grundidee von Greedy lässt sich aber auf viele andere Problemstellungen übertragen.

Unter anderem so ziemlich auf alles, was man als Einplanungsproblem in der Zeit oder im Raum verstehen kann.

Sukzessive Auswahl ist ja eine Folge von Entscheidungen. Die angekündigte Verallgemeinerung von Greedy ist nun, dass eine Lösung eben durch eine Folge von Entscheidungen zustande kommt.

Das heißt also, in jedem Schritt immer die nächste anstehende Entscheidung bestmöglich treffen – und nie revidieren.

Aber weiterhin nur kurzsichtig, keine Vorausschau auf weitere zu treffende Entscheidungen.

======

Diese Kurzsichtigkeit des Greedy-Ansatzes hat einen offensichtlichen Nachteil: Irgendwann mitten im schrittweisen Entscheidungsprozess wird sich vielleicht herausstellen, dass die bisher getroffenen Entscheidungen doch nicht so gut waren, wie es in den Momenten, in denen die Entscheidungen getroffen wurden, aussah. Schlimmstenfalls geht es ab einem gewissen Punkt einfach nicht mehr weiter.

Ich nehme wieder das Beispiel Stundenplan. Stellen Sie sich konkret vor, dass in vorherigen Entscheidungen viele Vorlesungen im Audimax so gelegt wurden, dass Montag und Dienstag voll sind, und jetzt wollen wir eine weitere Vorlesung im Audimax einplanen, die aufgrund anderer Restriktion nur montags oder dienstags einplanbar ist.

Backtracking erweitert Greedy dahingehend, dass die vorherigen Entscheidungen teilweise wieder rückgängig gemacht werden, wenn nötig. Man muss soweit zurückgehen, bis der Engpass aufgelöst wird. Der Standardansatz für Backtracking macht *alle* Entscheidungen rückgängig bis zu einem gewissen früheren Zeitpunkt. Das kann man natürlich variieren, denn nicht alle betroffenen Entscheidungen haben ja zum Engpass beigetragen. Im konkreten Beispiel brauchen Belegungen mittwochs bis freitags nicht unbedingt rückgängig gemacht zu werden.

Nachdem Entscheidungen rückgängig gemacht worden sind, muss natürlich dafür gesorgt werden, dass die nochmals zu treffenden Entscheidungen nicht hundertprozentig genauso wie beim ersten Mal getroffen werden, denn sonst würde der Algorithmus ja nochmals in genau denselben Engpass laufen.

Vorhin hatten wir zweistufige Ansätze. Die Idee von Backtracking lässt sich darauf übertragen. Das heißt, wenn sich in der zweiten Stufe herausstellt, dass keine akzeptable Lösung gefunden werden kann, dann geht der Algorithmus in die erste Stufe zurück, ändert das Ergebnis der ersten Stufe an kritischen Punkten ab und versucht dann auf dieser abgeänderten Basis nochmals, in der zweiten Stufe eine Lösung zu finden.

Wie bei den anderen hier betrachteten algorithmischen Konzepten könnte man auch über Backtracking noch viel mehr sagen. Aus Zeitgründen muss hier ein erstes Hineinschnuppern genügen.

======

Nun ein ganz anderer Ansatz. Lokale Suche sieht auf den ersten Blick so aus, als wäre das noch einmal dasselbe wie Greedy, aber das ist nicht korrekt. Der Unterschied ist, dass Greedy eine einzelne Lösung schrittweise aufbaut, also zwischendrin auf partiell konstruierten Lösungen arbeitet, während lokale Suche eine Sequenz von vollständig konstruierten Lösungen durchläuft.

Wir nehmen wieder das TSP als Beispiel, das heißt, gegeben sind endlich viele Objekte, die in beliebig definierter Distanz zueinander stehen. Gesucht ist eine zyklische Reihenfolge der Objekte, so dass die Summe der durchlaufenen Distanzen möglichst gering ist.

======

Benötigt für lokale Suche wird eine *Nachbarschaftsregel*, mit der man eine gegebene zulässige Lösung durch einfache Operationen in eine andere, sehr ähnliche, sozusagen *benachbarte* zulässige Lösung überführt. Für das TSP betrachten wir hier nur eine einzige, recht einfach zu implementierende, häufig verwendete, in der Praxis bewährte Nachbarschaftsregel. Dazu werden zuerst zwei beliebige Kanten aus der Rundtour herausgenommen, so dass zwei Pfade stehenbleiben.

======

Dafür werden dann zwei neue Kanten eingefügt, die die beiden Pfade wieder zu einer Rundtour zusammenfügen. Beachten Sie, dass einer der beiden Pfade dafür umgedreht werden muss. Wenn man entscheiden hat, welcher der beiden Pfade umgedreht wird, bleibt für die Auswahl der beiden einzufügenden Kanten nur noch eine einzige Möglichkeit.

Bei der lokalen Suche werden die Modifikationen immer so gewählt, dass die neue Lösung besser als die alte ist. Im konkreten Beispiel ist also die neue Rundtour kürzer als die alte. Ein wichtiger Punkt: Sollten die Distanzen nicht symmetrisch sein, das heißt, sollte die Distanz von A nach B nicht zwingend identisch mit der Distanz von B nach A sein, dann muss natürlich die Länge des *umgedrehten* Pfades für die Berechnung der neuen Rundtourlänge verwendet werden.

Die Suche fängt mit einer beliebigen Startlösung an und hört auf, wenn keine Verbesserung durch Anwendung der Regel mehr möglich ist. Die *letzte* gefundene Lösung ist natürlich die *beste* gefundene Lösung und wird daher als Ergebnis der lokalen Suche zurückgeliefert. Das ist aber im allgemeinen keine optimale Lösung, wenn es dumm läuft, kann diese Lösung sogar sehr schlecht sein. Raffinierte Variationen der lokalen Suche erlauben auch Verschlechterungen nach festen Regeln, um durch kurzfristige Verschlechterungen langfristig zu noch besseren Lösungen zu kommen. Dazu aber mehr in weiterführenden Veranstaltungen.

======

Noch ein anderes Beispiel für lokale Suche. Beim Rucksackproblem sind bekanntlich endlich viele Objekte jeweils mit einem Gewicht und einem Profit sowie ein maximales Gesamtgewicht gegeben, und gesucht ist eine Auswahl aus den Objekten, die das maximale Gesamtgewicht nicht überschreitet und den Gesamtprofit unter dieser Bedingung maximiert.

Wir betrachten zuerst eine schlechte Wahl der Nachbarschaft, danach eine bessere. Die schlechte Nachbarschaftsregel ist ganz einfach: Wähle ein Objekt aus, das in der momentanen Lösung drin ist, und entferne es. Wähle ein zweites Objekt aus, das in der momentanen Lösung *nicht* drin ist, und füge es ein. Also ein Austausch zweier Objekte.

Warum ist das schlecht? Weil die Anzahl der ausgewählten Objekte sich nicht ändern kann. Wenn die einzelnen Objekte sehr unterschiedliche Gewichte und Profite haben, können wir aber vorher kaum abschätzen, wie viele Objekte in einer sehr guten oder gar optimalen Auswahl drin sein würden.

Wie können wir dieses Problem möglichst einfach überwinden?

Indem wir zusätzlich erlauben, dass ein Objekt in die Lösung hinein oder aus der Lösung herausgeht, ohne dass ein anderes Objekt an dessen Stelle heraus- beziehungsweise hineinkommt.

Austausch eines Objekts mit einem anderen ist natürlich weiterhin eine sehr gute Möglichkeit, sollte man nicht abschaffen, aber es ist eben nicht mehr die *einzige* Möglichkeit.

======

Hier noch einmal die lokale Suche im Überblick.

Wie gesagt, beginnen wir mit einer beliebigen Startlösung. Die Startlösung kann sehr smart oder auch blind zufällig konstruiert sein. Oder wenn in der Praxis schon eine Lösung verwendet wird, kann man auch diese als Startlösung verwenden.

In jeder Iteration schauen wir, ob wir durch Anwendung der Nachbarschaftsregel zu einer besseren Lösung kommen. Hier gibt es verschiedene Möglichkeiten. Man kann die Laufzeit pro Iteration möglichst gering halten, indem man die erste bessere Lösung akzeptiert, die man in der Nachbarschaft von groß-S gefunden hat. Oder man kann die Anzahl Iterationen möglichst gering halten, indem man in jeder Iteration die wirklich beste Lösung sucht. Als Faustregel wird man letzteres aus offensichtlichen Gründen eher machen, wenn die Nachbarschaft klein ist, ersteres eher, wenn die Nachbarschaft groß ist. Einen Kompromiss aus beidem kann man sich natürlich auch überlegen.

Wenn wir in einer Iteration nicht mehr weiterkommen, ist die Suche natürlich zu Ende. Wie gesagt, da die durchlaufenen Lösungen immer nur besser werden, ist die letzte gesehene Lösung die beste und wird daher zurückgeliefert.

Wie Sie sicher schon bemerkt haben, ist mit groß-S immer die zuletzt erreichte Lösung benannt. Am Anfang ist *S* eben die Startlösung, und bei jedem Schritt vorwärts wird *S* die jeweils neue Lösung.

======

Wie schon angedeutet, gibt es viele Variationen dieser einfachen lokalen Suche. Wir betrachten hier nur die allereinfachste, bei der das Verhältnis zwischen Zeitaufwand und Ergebnis erfahrungsgemäß aber auch schon recht gut sein kann.

Die Variation besteht einfach darin, dass lokale Suche nicht nur einmal, sondern mehrmals angewandt wird.

Je nachdem, wie viel Laufzeit man sich leisten kann und will, gerne ein paar tausend Male oder sogar ein paar Millionen Male.

Natürlich nicht immer mit derselben Startlösung, sondern die Startlösung wird systematisch so variiert, dass sie aus möglichst allen Bereichen des Lösungsraums mehr oder gleichwahrscheinlich gewählt wird. Am einfachsten geht das natürlich, wenn man die Startlösung rein durch Zufallsentscheidungen konstruiert.

Zum Beispiel beim TSP könnte man von einer leeren Lösung ausgehen und wiederholt eine Kante mit uniformer Wahrscheinlichkeit zufällig wählen, und wenn diese Kante bis dahin noch nicht hinzugefügt war und auch nicht mit den vorher hinzugefügten Kanten eine Rundtour auf einer echten Teilmenge der Punkte schließt, dann wird sie hinzugefügt. Das wird solange wiederholt, bis eine Rundtour zustande gekommen ist. Auf diese Weise ist jede mögliche Rundtour gleichwahrscheinlich.

Natürlich könnte man zweimal so dieselbe Startlösung generieren und damit einen Versuch verschwenden. Aber wenn das zu lösende Problem nicht gerade winzig ist, dann ist die Wahrscheinlichkeit dafür astronomisch gering.

Man kann die Prozedur noch verbessern, wenn man sich während der wiederholten lokalen Suche merkt, was die guten bisher gefundenen Lösungen gemeinsam haben. Zum Beispiel beim TSP könnte man sich zu jeder Kante merken, wie gut die bisher gefundenen Lösungen durchschnittlich waren, die diese Kante enthalten haben. Die Kanten, bei denen dieser Wert hoch ist, bekommen dann bei weiteren Wiederholungen eine höhere Auswahlwahrscheinlichkeit.

======

Zum Abschluss noch ein ganz genereller Punkt. Eine gute Möglichkeit für effektive algorithmische Ansätze wird häufig übersehen: dass man nicht einfach wie bisher beschrieben einen generisch verwendbaren Algorithmus einfach so anwendet, sondern dass man genauer in die algorithmische Problemstellung hineinschaut und versucht, charakteristische Merkmale zu identifizieren und auszunutzen. Wir betrachten konkret diese drei aussichtsreichen Ideen auf den folgenden Folien.

======

Für den ersten Punkt, Granularität der Daten, nehmen wir wieder einmal einen alten Bekannten als Beispiel her, das Rucksackproblem. Hier noch einmal formal definiert, weil wir im Folgenden diese Notationen brauchen.

Ein Input hat mehrere Bestandteile.

Wir können nur ein gewisses maximales Gesamtgewicht einpacken.

Wir wollen eine Auswahl aus insgesamt n Objekten einpacken.

Und jedes dieser Objekte hat ein Gewicht, und wenn wir ein Objekt mitnehmen, erlösen wir daraus einen spezifischen Profit.

Der Output ist eben die Auswahl der Objekte, die wir dann tatsächlich einpacken.

Natürlich müssen wir darauf achten, dass unsere Auswahl nicht das zulässige Gesamtgewicht überschreitet.

Aber unter allen möglichen Auswahlen, die das zulässige Gesamtgewicht nicht überschreiten, möchten wir natürlich eine Auswahl berechnen, so dass die Summe der Profite aus den einzelnen ausgewählten Objekten möglichst hoch ist.

======

Wir brauchen auf dieser Folie noch ein paar Vorarbeiten, erst auf der nächsten Folie kommen wir zu dem Thema, um das es hier eigentlich geht, Granularität. Wir stellen erst einmal fest, dass die optimale Auswahl rekursiv formuliert werden kann.

Entweder ist das Objekt Nummer n *nicht* in der optimalen Auswahl, dann ist die optimale Auswahl aus 1 bis n identisch mit der optimalen Auswahl aus 1 bis n minus 1.

Oder das Objekt Nummer n ist eben *doch* in der optimalen Auswahl.

Dann besteht die optimale Auswahl also aus dem Objekt Nummer n und einer gewissen Auswahl aus den Objekten 1 bis n minus 1.

Und letztere Auswahl lässt sich exakt charakterisieren, nämlich eine optimale Auswahl aus 1 bis n minus 1 für das restliche Gesamtgewicht, das übrigbleibt, wenn wir das Gewicht von Objekt Nummer n vom maximal zulässigen Gesamtgewicht abrechnen.

======

Es wäre wieder nicht besonders geschickt, diese Rekursionsgleichung wirklich als einen rekursiven Algorithmus zu implementieren, denn das würde auf eine exponentielle Anzahl von rekursiven Aufrufen hinauslaufen. Statt dessen wenden wir wieder das Prinzip der dynamischen Programmierung an, das wir vorhin schon kennen gelernt haben.

Hier kommen wir jetzt endlich zu dem Thema, um das es geht, Granularität der Daten. Wir machen natürlich keinen ernstzunehmenden Fehler, wenn wir das Gesamtgewicht und die Gewichte der einzelnen Objekte nicht als beliebige reelle Zahlen definieren, sondern als ganzzahlige Vielfache einer ausreichend kleinen Gewichtseinheit. Dadurch können wir eine Matrix aufstellen, deren Spalten allen möglichen Gewichtswerten von null bis zum maximalen Gesamtgewicht entsprechen.

Zum Beispiel, wenn es wirklich um das Packen eines Rucksacks geht, dann reicht es sicherlich, die Gewichte der einzelnen Objekte auf zehn Gramm oder meinetwegen ein Gramm genau zu nehmen. Oder ein anderes Beispiel: Beim Möbeltransport mit einem Möbelwagen sollte es normalerweise ausreichen, alle Möbel auf ein Kilogramm oder meinetwegen auf hundert Gramm genau zu wiegen.

======

Im Matrixeintrag für Zeile i und Spalte j soll dann der optimale Gesamtprofit stehen über alle Auswahlen aus den Objekten 1 bis i, so dass das Gewicht j nicht überschritten wird. Der Profit ist natürlich null, wenn entweder i null ist, also keine Objekte zur Auswahl zugelassen sind, oder wenn das maximale Gesamtgewicht null ist.

Das ist der erste Fall der Rekursionsgleichung, also wenn die optimale Auswahl aus den Objekten 1 bis i identisch mit der optimalen Auswahl aus 1 bis i minus 1 für dasselbe maximal zulässige Gesamtgewicht ist.

Dieses Ergebnis wird natürlich überschrieben, wenn der zweite Fall der Rekursionsgleichung besser ist. Der zweite Fall kann natürlich nur eintreten, wenn das maximale Gesamtgewicht j nicht geringer als das Gewicht des Objekts Nummer i ist.

Neben dem optimalen Profit können wir immer auch die zugehörige optimale Auswahl aus der Matrix zurückrechnen, wenn wir uns in jedem Matrixeintrag noch zusätzlich merken, welcher der beiden Fälle der Rekursionsgleichung hier der bessere war. Vom Matrixeintrag für klein-n und groß-G können wir dann schrittweise zurückgehen bis null-null, indem wir in einer Zeile i und einer Spalte j Folgendes machen: Wenn der erste Fall der Rekursionsgleichung für das Paar (i,j) der bessere war, dann gehen wir genau eine Zeile höher. Ansonsten gehen wir statt dessen so viele Spalten nach links, wie das Objekt Nummer i schwer ist. Im letzteren Fall gehört das Objekt Nummer i zur Auswahl dazu, im ersteren Fall nicht.

======

Dieses Beispiel soll hier reichen zum ersten Thema, Granularität der Daten. Nun also zum zweiten charakteristischen Merkmal.

Wir schauen uns dazu ein generisches Beispiel an, nämlich alle Anwendungsfälle, in denen die Kanten eines Netzwerks zu unterschiedlich hohen Kategorien gehören.

Zum Beispiel im deutschen Straßennetz bilden die Autobahnen die höchste Kategorie, gefolgt von Bundesstraßen, dann normale Landstraßen und Durchgangsstraßen, schließlich Nebenstraßen. Und im deutschen Bahnverkehr sind die ICEs die Züge höchster Kategorie, dann die ICs, dann Interregio und Regionalexpress, dann Regionalbahnen, schließlich Nahverkehr wie etwa S-Bahnen. Sowohl bei Straßenverkehr wie auch bei schienengebundenem Verkehr ist die Situationen in anderen Ländern natürlich analog, nur die Details sind andere.

Offensichtlich sieht die überwiegende Mehrzahl aller sinnvollen längeren Verbindungen so aus, dass die Kategorien vom Startpunkt aus zuerst strikt aufsteigend durchlaufen werden bis zur obersten oder auch vielleicht nur bis zur zweit- oder drittobersten Kategorie, aber eben strikt aufsteigend.

Und danach werden sie bis zum Endpunkt der Reise noch einmal absteigend durchlaufen. Solche Verbindungen nennt man *bitonisch* – biton statt monoton.

Ein klassischer Fall, wo die Betrachtung bitonischer Verbindungen nicht ausreicht, ist Bahnfahren in Frankreich. Man kommt mit einem Zug von hoher Kategorie an einem der großen Pariser Kopfbahnhöfe an, fährt dann mit der Metro zu einem anderen Kopfbahnhof und steigt dort wieder in einen Zug von hoher Kategorie.

Fälle, wo bitonische Verbindungen nicht ausreichen, lassen sich gut vorab identifizieren und vor allem *lokalisieren*, wie etwa das Beispiel Pariser Kopfbahnhöfe. Diese Fälle können schon bei der Datenaufbereitung gefunden und in die eigentliche Verbindungssuche geeignet eingearbeitet werden. Die Verbindungssuche sucht ansonsten strikt nur bitonische Verbindungen, was die zu durchsuchenden Verbindungsmöglichkeiten natürlich drastisch reduziert.

======

Ein weiteres Beispiel rund um Netzwerke, die in großer Fläche ausgelegt sind. Grob gesprochen, ist ein in der Ebene ausgelegtes Netzwerk planar, wenn es keine Brücken und Tunnel gibt, sondern jedes Zusammentreffen von Kanten ist ein Kreuzungspunkt.

Davon weichen reale Netzwerke natürlich häufig ab, aber nur ein wenig, das heißt, verglichen mit der Gesamtzahl an Knoten und Kanten gibt es nur sehr wenige Brücken und Tunnel, und diese Brücken und Tunnel sind auch sehr kurz und überbrücken beziehungsweise untertunneln nur wenige andere Kanten. Der planare Grundcharakter des Netzwerks bleibt erhalten.

Dadurch macht es Sinn, die geographischen Koordinaten auszunutzen. Ein einfaches Beispiel, das vor langer Zeit tatsächlich für die Berechnung von Zugverbindungen benutzt wurde, sehen Sie auf den nächsten beiden Folien.

======

Grundidee ist, eine Ellipse über die Ebene zu legen, deren Brennpunkte gerade Start und Ziel der Suche sind, und nur innerhalb dieser Ellipse Verbindungen zu suchen.

======

Das klappt natürlich nicht immer, wie dieses Beispiel zeigt. Deshalb war man ziemlich bald dazu übergegangen, wichtige Bahnhöfe in der Nähe der Ellipse zusätzlich in die Suche aufzunehmen. Das reduziert die zu durchsuchende Datenmenge in den meisten Fällen immer noch gewaltig.

======

Ganz zum Schluss noch die dritte und letzte von uns hier betrachtete Idee zum Thema Datenprofil ausnutzen: zusätzliche Informationen.

Zum Beispiel bei Anwendungen des Rucksackproblems kommt es häufig vor, dass einzelne Objekte schon vorab gesetzt sind, also auf jeden Fall mitgenommen werden sollen. Oder dass ein bestimmtes Objekt mitzunehmen nur Sinn macht, wenn auch ein bestimmtes anderes Objekt mitgenommen wird, Zeltplane beispielsweise nur dann, wenn auch die zugehörigen Heringe mitgenommen werden. Oder beim TSP: Häufig gibt es Vorschriften der Art, dass ein bestimmter Punkt vor einem gewissen anderen Punkt zu durchlaufen ist.

Bei algorithmischen Problemstellungen, in denen im weiteren Sinne etwas zu rekonstruieren oder zu entschlüsseln oder zu übersetzen ist, ist eher nichts vorgeschrieben, aber häufig sind zusätzliche Informationen aus anderen Quellen bekannt. Wenn es beispielsweise darum geht, von Menschen geschriebene Texte zu verstehen, also die Bedeutung eines Textes zu rekonstruieren, dann kann man häufig auf bibliographische Informationen, Verschlagwortungen und ähnliches zurückgreifen.

Letztendlich laufen die verschiedenen Wege, zusätzliche Informationen auszunutzen, auf zwei verschiedene Grundprinzipien hinaus.

Naheliegend ist es natürlich, solche Zusatzinformationen direkt in den Algorithmus zu integrieren. Zum Beispiel beim Greedy-Algorithmus kann man alle gesetzten Objekte im Rucksackproblem einfach schon einmal vorab in die Lösung einfügen. Und wenn im TSP ein Punkt unbedingt vor einem anderen kommen soll, fügt der Greedy-Algorithmus eben keine Kante ein, durch die ein Pfad vom zweiten zum ersten Punkt geschlossen würde. In einer lokalen Suche hingegen würden Modifikationen verboten werden, durch die im Rucksackproblem vorgeschriebene Objekte aus der Lösung herausgenommen werden oder durch die im TSP eine falsche Reihenfolge zweier Punkte entstehen würde.

Das zweite Grundprinzip ist anwendbar, etwa wenn statistische Methoden eingesetzt werden, um etwas zu rekonstruieren, zu entschlüsseln oder zu übersetzen oder ähnliches. Beispielsweise setzt man statistische Verfahren ein, um die Bedeutung eines Wortes in einem Text zu bestimmen, denn ein und dasselbe Wort kann in verschiedenen Kontexten ja ganz unterschiedliche Bedeutungen haben. Ein gerne verwendetes Beispiel ist das Wort Bank im Deutschen, das einerseits eine Bank zum Sitzen, andererseits ein Finanzinstitut bezeichnet.

Bei vielen Texten sind die Wortbedeutungen durch viel Fleißarbeit von Annotatoren bekannt. Man kann nun auszählen, wie häufig das Wort Bank zusammen mit bestimmten anderen Wörtern im Sinne von Sitzbank beziehungsweise im Sinne von Finanzbank verwendet wird. Zum Beispiel wird das Wort Bank eher eine Sitzbank meinen, wenn im Text drumherum, also im *Kontext*, Wörter wie Baum und Rasen auftreten, und eher eine Geldbank, wenn Wörter wie Kredit und Aktien zu finden sind.

Das ergibt dann eine Wahrscheinlichkeit für jede der beiden Bedeutungen in einem gegebenen Kontext. Natürlich ist das nur die Grundidee. Im *maschinellen Lernen*, einer Forschungsdisziplin der Informatik, wird diese Grundidee raffiniert weiterentwickelt.

Damit sind wir mit dem Thema Datenprofil, und was diese Lehrveranstaltung angeht, auch mit dem gesamten Thema Algorithmische Konzepte durch.

======