Datenstrukturen & Algorithmen 16. Mai 2016

Handout 11

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Spielbaumsuche, Backtracking, Branch & Bound, Online Algorithmen, Approximations-Algorithmen

Links

- Min-Max
  http://en.wikipedia.org/wiki/Minimax (http://goo.gl/oqnf)
- Branch & Bound
  http://en.wikipedia.org/wiki/Branch_and_bound (http://goo.gl/iJXCx)
- Backtracking
  http://en.wikipedia.org/wiki/Backtracking (http://goo.gl/mlGa)
- Jeff Ericksons Notizen zu Backtracking
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/03-backtracking.pdf
  (http://goo.gl/L0ZA8h)
- Jeff Ericksons Notizen zu Approximations Algorithmen
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/31-approx.pdf
  (http://goo.gl/AZUfsd)

Backtracking

Ein Backtracking (Exhaustive Search) Algorithmus versucht eine Lösung schrittweise (inkrementell) aufzubauen. Wann immer der Algorithmus sich zwischen mehreren Alternativen für den nächsten Schritt entscheiden muss, probiert er *alle* möglichen Optionen (oft rekursiv) aus. Der Algorithmus verwirft eine Teillösung, wenn er erkennt, dass keine gültige Lösung mehr erhalten werden kann. Dazu muss er bisher gemachte Entscheide rückgängig machen (*backtracking*). Der Backtracking Algorithmus verfährt nach einer systematischen Erzeugungsordnung (*generation order*), so dass keine Lösung übersehen oder mehrfach betrachtet wird.

Konzeptuell können die möglichen Teillösungen als Knoten eines Baumes gesehen werden (Suchbaum). Jede potentielle Teillösung ist Vater der Teillösungen, die aus dieser hervorgehen durch einen Schritt. Die Blätter dieses Baumes sind (Teil)lösungen, die nicht mehr erweitert werden können. Der Backtracking Algorithmus durchsucht diesen Suchbaum rekursiv (oft in DFS - order): Bei jedem Knoten überprüft der Algorithmus, ob die Teillösung (überhaupt noch) zu einer validen Lösung vervollständigt werden kann. Wenn nicht, wird der gesamte Teilbaum ausgelassen (*pruned*).

Beispiel: Das n Damen Problem

Ein klassisches Problem ist das n Damen Problem, welches erstmals vom deutschen Schach Enthusiasten Max Bezzel 1848 für das Standard 8×8 Brett präsentiert wurde. 1850 ist es durch Franz Nauck gelöst und verallgemeinert worden für grössere Spielbretter. Das Problem ist n Damen auf einem $n \times n$ Schachbrett zu positionieren, so dass sich keine zwei Damen angreifen, d.h. keine zwei Damen dürfen in der gleichen Spalte, Zeile oder Diagonale stehen.

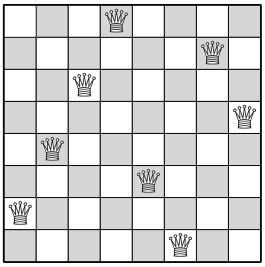


Abb.: Eine mögliche Lösung für das 8 Damen Problem $(4, 7, 3, 8, 2, 5, 1, 6)$

Da offensichtlich in jeder Lösung genau eine Dame in jeder Zeile steht, werden wir mögliche Lösungen durch ein Array $Q[1 \ldots n]$ repräsentieren. $Q[i]$ gibt dabei an, welches Feld auf der $i$ten Zeile eine Dame enthält.

Um eine Lösung zu finden, werden wir die Damen nun Zeile für Zeile platzieren (startend mit der ersten Zeile).

Um schnell zu überprüfen, ob eine Spalte oder Diagonale bereits belegt ist, werden wir folgende boolean Arrays betrachten:

- $column[1 \ldots n]$, dabei ist $column[i] =$ TRUE, falls in der $i$ten Spalte eine Dame platziert wurde
- $diag[1 \ldots 2n-1]$, dabei ist $diag[i] =$ TRUE, falls in der $i$ten links-rechts Diagonale eine Dame platziert wurde
- $diag2[1 \ldots 2n-1]$, dabei ist $diag2[i] =$ TRUE, falls in der $i$ten rechts-links Diagonale eine Dame platziert wurde

Der folgende rekursive Algorithmus zählt rekursiv alle möglichen Lösungen auf, die konsistent sind mit einer gegebenen Teillösung. Der Eingabeparameter k ist die erste freie Zeile.

```
RECURSIVE-N-QUEENS(Q, k)
 1  if k == n + 1
 2      output Q
 3  else for j = 1 to n // mögliche Spalte für kte Dame
 4          // gültige Position?
 5          if spalte[j] == FALSE and diag[k + j] == FALSE and diag2[n − k + j] == FALSE
 6          // setzte Dame
 7              Q[k] = j
 8              spalte[j] = TRUE, diag[k + j] = TRUE, diag2[n − k + j] = TRUE
 9              RECURSIVE-N-QUEENS(Q, k + 1)
10              // mache Zug rückgängig
11              spalte[j] = FALSE, diag[k + j] = FALSE, diag2[n − k + j] = FALSE
```

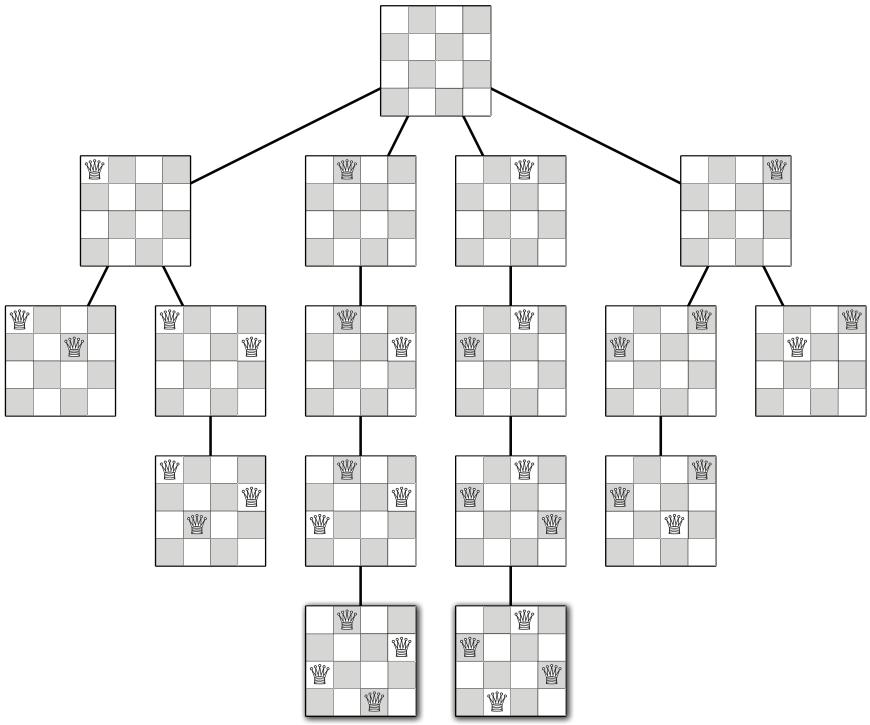


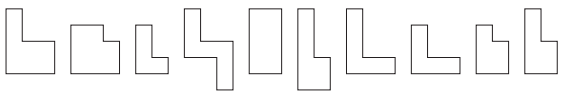
Abb.: Der vollständige Suchbaum für das 4 Dame Problem.

Anmerkung

Eine *explizite* Lösung für das n Dame Problem ist bekannt. Vergleiche hierzu *Explicit solutions to the N-queens problem for all N* (`http://goo.gl/JOQum`). Zugreifbar aus dem ETH Netz.

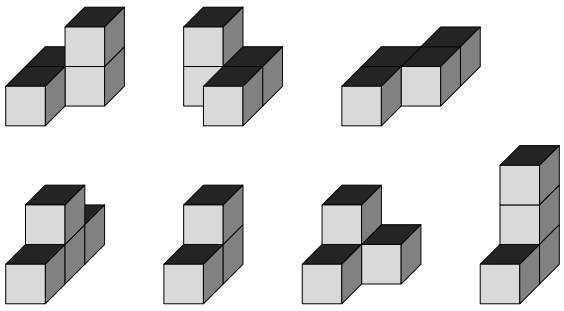
Stop and Think

- Verwende Backtracking um die Anzahl Lösungen für das 5 Damen Problem zu zählen. Wieviele davon sind nicht isomorph?
- Betrachte ein Puzzle mit den folgenden Teilen, die auf einem 8x8 Brett angeordnet werden müssen.



  Schreibe ein Backtracking Programm um die Anzahl verschiedener nicht-isomorphen Lösungen zu finden.

- Das Soma Würfel Puzzle ist es folgende sieben Teile in einem 3x3x3 Würfel anzuordnen.



  Schreibe ein Backtracking Programm um die 240 nicht-isomorphen Lösungen zu generieren.

Branch & Bound

Branch & Bound ist eine Beschleunigungs-Methode für Algorithmen für Optimierungsprobleme. Branch & Bound basiert auf *Backtracking*, das eine *erschöpfende* Suchtechnik im Raum der möglichen Lösungen ist. Das Problem dabei ist, dass die Anzahl der möglichen Zustände (Grösse des Suchbaumes) oft exponentiell oder von der Grössenordnung $2^n, n!, n^n \ldots$ in der Eingabegrösse n sein kann.

Branch & Bound ist eine Methode, um in einem grossen Entscheidungsbaum ganze Teilbäume mittels Schranken (*Bounds*) abschneiden zu können. Grob gesagt ist Branch & Bound eine Technik um Backtracking zu beschleunigen, indem Teile des Suchraumes nicht durchsucht werden, weil man erkennt dass *diese Bereiche keine optimale Lösung enthalten können.*

Dies wurde in gewisser Art schon in obiger Backtracking-Lösung genutzt, in dem wir schnell Teillösungen verworfen haben, die nicht mehr zu einer gültigen Gesamtlösung erweitert werden können (da sie mehr als eine Dame in der gleichen Zeile, Spalte oder Diagonalen enthalten würden).

Ein Schritt in einen Branch & Bound Algorithmus besteht aus zwei Teilen:

- **Bound:** Anfangs wird überprüft, ob die aktuelle Teillösung bereits teurer ist als die bisher beste gefundene vollständige Lösung. Falls ja, kann abgebrochen werden. Dazu muss es natürlich möglich sein, dass bereits die Kosten einer noch unvollständigen Lösung ermittelt werden können. Dazu können wir uns obere und untere Schranken der Lösung überlegen. Der Gedanke dahinter ist, dass oft bereits an "billigen" Teillösungen zu erkennen ist, ob sie sich überhaupt noch zu einer guten Lösung vervollständigen lassen. Bei der Entwicklung guter Schranken ist Ideenreichtum gefragt. Die Berechnung darf dabei auch nicht zu lange dauern.
- **Branch:** Ist man bei einer vollständigen Lösung angelangt, wird diese Lösung mit der bisher besten verglichen und evtl. übernommen. Sonst wird über alle möglichen Erweiterungen der aktuellen Teillösung iteriert und jeweils zum nächsten Schritt übergegangen.

Je nach dem wie eng die Schranken gesetzt werden, kann die Laufzeit zum Finden einer optimalen (oder genügend guten) Lösung stark verkürzt werden. Branch & Bound wird auch häufig in Zusammenhang mit Heuristiken gebraucht. Oft wird auch eine Schranke für die Kosten der optimalen Lösung vorberechnet.

Typische Anwendungsbeispiele sind Maschinenbelegungsprobleme, das Knapsack Problem oder Rucksackproblem, das als Teilproblem in grösseren Optimierungsproblemen auftreten kann, das Traveling Salesman Problem oder Rundreiseproblem und andere.

Spielbaumsuche

Spielbaum

Jedes Strategiespiel (2 Spieler, die abwechselnd ziehen; vollständiger Information; alle Partien endlich) hat einen endlichen *Spielbaum*. In einem Spielbaum entspricht die Wurzel der Ausgangsposition des Spiels und die inneren Knoten sind momentane Zustände einer Partie. Die Blätter repräsentieren die Endpositionen einer Partie. Folglich entsprechen die Kanten auf Level $2n + 1$ ($n \in \mathbb{N}$) Zügen von Spieler A und die Kanten auf Level $2n$ Zügen von Spieler B. Eine Partie entspricht einem Pfad von der Wurzel zu einem Blatt.

Strategie

Ein Spiel kann nun wie folgt gelöst werden: Jedem Blatt wird zugeordnet wer gewonnen hat (resp. ob unentschieden ist). Nun kann der Baum von den Blättern her ausgefüllt werden, so dass in jedem Knoten (jedem möglichen Spielzustand) steht wer gewinnen wird. Dabei wird wie folgt ausgefüllt:

- Falls Spieler A als nächstes zieht, dann wird in der aktuellen Konstellation Spieler A gewinnen, falls mindestens ein Zug von A zu einer Gewinnsituation für A führt.

- Falls kein Zug zu einer Gewinnsituation aber mind. ein Zug zu einer Remisituation führt, wird die aktuelle Konstellation auch eine Remisituation.
- Andernfalls wird es eine Gewinnsituation für Spieler B (= Verlustsituation für Spieler A).

Für Spieler B erfolgt die Einteilung analog.

Min-Max

Wir benennen nun die Spieler A und B um in Max und Min. Desweiteren sagen wir nicht mehr Gewinnsituation, sondern bewerten die Situation X mit

$$bew(X) = \begin{cases} 1 & \text{Gewinnsituation für Max (A)} \\ 0 & \text{Remisituation} \\ -1 & \text{Gewinnsituation für Min (B)} \end{cases}$$

Unsere Strategie-Regeln vereinfachen sich dann zu:
Max ist am Zug: wähle den Zug, der in einer Situation mit **maximaler Bewertung** endet.
Min ist am Zug: wähle den Zug, der in einer Situation mit **minimaler Bewertung** endet.

Feldbewertung

Dieses Vorgehen erlaubt es, jedes strat. Spiel optimal zu spielen. Jedoch ist in realen Spielen der Spielbaum zu gross (er wächst exponentiell in der Länge der Partien). Darum wird nicht der ganze Spielbaum generiert, sondern (ausgehend von der aktuellen Spielposition) bis zu einer gewissen Tiefe simuliert. Dann wird die Simulation abgebrochen und die aktuelle Spielsituation lokal betrachtet und bewertet (kleine Werte sind gut für Min, grosse gut für Max). Danach wird ausgehend von diesen Werten von unten her der jeweils beste Zug für Min und für Max in jeder simulierten Situation bestimmt (vergleiche Abbildung).

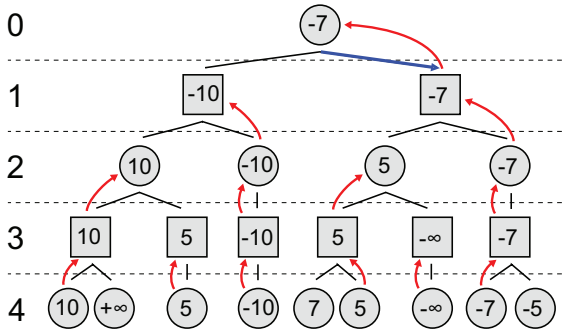


Abb.: Min-Max Algorithmus. Max ist am Zug.
Der Min-Max Algorithmus schlägt den zweiten Zug vor.

Alpha-Beta-Pruning

Im Min-Max Algorithmus tritt oft eine Situation analog zu folgender Abbildung unten links auf: Spieler Max lässt gerade seine zweite Zugsmöglichkeit simulieren (die in 4 enden wird). Er weiss dabei schon, dass der erste Zug auf eine Spielsituation mit der Bewertung 5 fführen wird. Nachdem Spieler Max seinen zweiten Zug simuliert hat, wird Spieler Min seine Antwort-Züge simulieren. Der zweite mögliche Zug für Min gibt hier 4 zurück

Ab diesem Zeitpunkt ist es Max egal, was die anderen Züge von Min zurückgeben werden! Denn auch wenn diese extrem gross wären, so wird Min (der ja die Werte minimiert) höchstens eine Spielposition mit der Bewertung 4 wählen (potentiell wird dieser Wert noch kleiner). Max hat aber schon einen Zug zur Verfügung, der den Wert 5 zurück gab (sein erster Zug), folglich wird er den zweiten Zug sowieso nicht wählen und ist nicht an seiner weiteren Evaluierung interessiert.

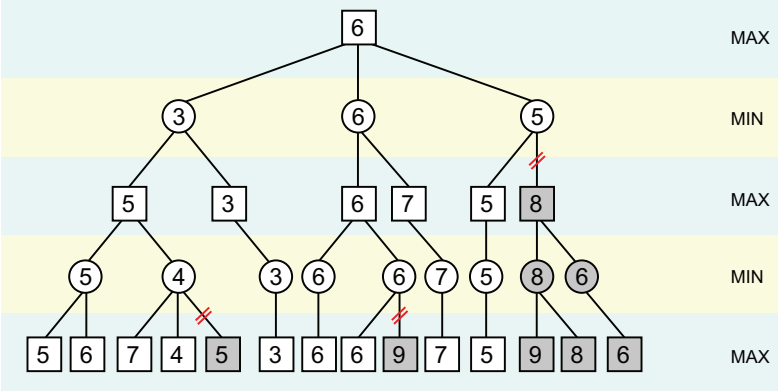


Abb.: $\alpha - \beta$ pruning

In solchen Situationen können ganze Teilsuchbäume übersprungen werden, die Suche wird effizienter. Dabei ist zu bemerken, dass der Algorithmus immer noch optimal ist. Er wird in jedem Fall das gleiche Ergebnis liefern, wie der nicht-optimierte Min-Max-Algorithmus.

```
/*
* Aufruf : alphabeta (0, MAX oder MIN , -INFINITY , INFINITY );
* Die Zugauswahl wird in der Implementation weggelassen
* (also das Speichern des besten Zuges auf Tiefe 0)
*/
 double alphabeta (int tiefe, char spieler, double alpha, double beta)
 {
        // die Bewertung der aktuellen Situation
        double bewertung;

        if (tiefe >= MAXTIEFE)
                return bewerteSituation();

        // betrachte alle Zuege
        foreach (Zug z) {
                z.simuliere;
                if (gewonnen)
                        bewertung = SIEG - tiefe;
                else
                        bewertung = alphabeta(
                            tiefe+1, gegner(spieler), alpha, beta);
                z.rueckgangig;

                if (spieler == MAX) { // MAX maximiert alpha
                        alpha >?= bewertung; // neuer Zug besser?
                        if (alpha >= beta) // Beta-Cut?
                                break;
                }
                else { // MIN minimiert beta
                        beta <?= bewertung; // neuer Zug besser?
                        if (beta <= alpha) // Alpha-Cut?
                                break;
                }
        }

        if (spieler == MAX)
                return alpha;
        else
                return beta;
 }
```

Anmerkungen

- *Dame*: *Chinook* (http://de.wikipedia.org/wiki/Chinook_(Programm)) beendete 1994 die 40-jährige Vorherrschaft von World-Champion Marian Tinsley. Das Programm benutzte

eine Endspieldatenbank für alle Positionen mit 8 oder weniger Steinen auf dem Feld. Im Ganzen beinhaltete die Endspieldatenbank deshalb 443,748,401,247 Positionen.

- *Schach*: *Deep Blue* (http://en.wikipedia.org/wiki/Deep_Blue_(chess_computer)) besiegte 1997 World Champion Gary Kasparov in einem 6 Spiele-Match. Deep Blue durchsucht 200 Millionen Positionen pro Sekunde, benutzt eine äusserst durchdachte Bewertefunktionen und beinhaltet (nicht veröffentlichte) Methoden die es erlauben bis zu einer Tiefe von 40 Zügen zu suchen.
- *Othello*: Menschliche Champions verweigern Spiele gegen Computerprogramme (die anscheinend zu gut sind)
- *Go*: Menschliche Champions haben lange Spiele verweigert gegen Computerprogramme. Diese waren zu schlecht. In Go gibt es über 300 Möglichkeiten pro Zug. Deshalb verwendeten die meisten Programme vorprogrammierte Muster oder Ähnliches. Google DeepMind's Programm AlphaGo hat im Oktober 2015 erstmals den mehrfachen Europameister Fan Hui besiegt. Es ist damit das erste Program, das unter Turnierbedingungen einen professionellen Go-Spieler bezwingen konnte. Im März 2016 gewann AlphaGo gegen Lee Sedol (9. Dan) in einem Match von fünf Runden mit 4:1. AlphaGo kombiniert geschickt Techniken des maschinellen Lernens (insbesondere Deep Learning, Reinforcement Learning) mit Elementen der Spielbaumsuche (z.B. Suche, Monte Carlo Rollouts).

Verbesserungen

- *Best First*: Um die Anzahl der Cutoffs zu erhöhen, können in jedem Level die Nachfolger bewertet werden und sie nach Bewertung abarbeiten (*best-first*-Strategie). Wenn der (lokal) beste Zug zuerst gewählt wird, erwartet man eine erhöhte Anzahl Cutoffs wenn die anderen Nachfolger durchsucht werden (think about it)
- *Iterative Deepening*: Die iterative Tiefensuche ist ähnlich wie die normale Tiefensuche: Iterativ wird eine beschränkte Suche durchgeführt (man sucht bis zu einer fixen Tiefe), und dabei wird das Level, bis zu welchem die Suche das Spiel erkundet, bei jeder Iteration um eins erhöht. Im ersten Schritt wird also ein Zug vorausgeschaut. Im nächsten Schritt wird dann 2 Züege weit erkundet. Dabei kann man nun zum Beispiel die Resultate einer Iteration brauchen um die Züge in der nächsten Iteration zu ordnen (vgl. Best First).
- *Zobrist-Hash*: Insbesondere bei Brettspielen ist es möglich (und in den meisten Szenarien sehr wahrscheinlich), dass es viele verschiedene Sequenzen von Zügen auf die gleiche Spielsituation führen. Dies führt dazu, dass wir in der Spielbaumsuche äquivalente Knoten haben. Überlege dir dies beispielsweise für das Tic-Tac-Toe-Spiel. Um eine Spielsituation/Knoten nicht mehr als einmal zu erkunden/durchsuchen, kann das Result gehasht werden. Dazu kann der *Zobrist-Hash* verwendet werden: Zobrist Hashing startet durch zufällig generierte Bitstrings für jedes mögliche Element eines Brettspiels. Für eine gegeben Spielsitutation werden die Bitstrings der einzelnen Steine/Elemente zusammengefasst mit einem bitweisen XOR. Wenn die Bitstrings lang genug sind, werden verschiedene Brettpositionen mit hoher Wahrscheinlichkeit auf unterschiedliche Werte gehasht.

  Für das Tic-Tac-Toe Spiel kann dies beispielsweise folgendermassen aussehen (think about it!)

```
#define EMPTY 0
#define CROSS 1
#define CIRCLE 2

int R[3][3][3];

void init() {
   srand(0);
   for (int x = 0; x < 3; ++x) {
      for (int y = 0; y < 3; ++y) {
         R[x][y][EMPTY] = rand();
         R[x][y][CROSS] = rand();
         R[x][y][CIRCLE] = rand();
      }
   }
```

```
}

int hash(int** game) {
    int key = 0;
    for (int x = 0; x < 3; ++x) {
        for (int y = 0; y < 3; ++y) {
            int whatIsAtXY = game[x][y];
            key ^= R[x][y][whatIsAtXY];
        }
    }
    return key;
}
```

Der Zobrist-Hash hat nun den Vorteil, dass wir Spielzüge leicht updaten können. Wenn wir den Hash der aktuellen Situation kennen und wir machen einen Zug (bei Tic-Tac-Toe: ein Kreuz/Kreis an eine leere Stelle schreiben) können wir den Hash-Wert der resultierenden Spielsituation inkrementell bestimmen.

Beispielsweise (Tic-Tac-Toe): das linke obere Feld ist leer und wir machen einen "Kreis" in dieses Feld. Wir mssen dann zwei Änderungen vornehmen. Zunächst müssen wir die "das linke obere Feld ist leer" - Information aus dem Schlüssel entfernen und dann dann die "Kreis im linken oberen Feld" Information hinzufügen. Hier nutzen wir die XOR-Operation um die Information zu updaten (again think about it)

```
/* ... */
// Strip the information of the empty tile.
int newKey = oldKey ^ R[0][0][EMPTY];

// Now encode the information on the circle being placed there.
newKey ^= R[0][0][CIRCLE];
/* ... */
```

vgl. http://en.wikipedia.org/wiki/Zobrist_hashing (http://goo.gl/w8Pn2)

Approximations-Algorithmen: Schnell, aber nicht optimal

Die NP-Vollständigkeit der zugehörigen Entscheidungsprobleme legt nahe, dass es keine Polynomialzeitalgorithmen für die Lösung vieler Optimierungsprobleme gibt (ausser P=NP). Trotzdem möchten wir diese hartnäckigen Probleme bearbeiten. Es gibt verschiedene Möglichkeiten, solche Probleme zu vereinfachen

- **Super-polynomieller Aufwand:** Es kann sein, dass wir nicht mehr erfordern, dass ein Algorithmus in polynomieller Zeit läuft. In manchen Fällen existieren Algorithmen die knapp super-polynomiell laufen und vernünftig schnell in praktischen Anwendungen sind. Techniken wie Branch-and-Bound oder Dynamic Programming können hier nützlich sein. Das *Rucksack Problem* ist beispielsweise NP-vollständig, aber gilt als "einfach", da ein "pseudo-polynomieller" Algorithmus existiert.
- **Probabilistische Analyse von Heuristiken:** Eine andere Möglichkeit ist es, die Bedingung zu lockern, dass der Algorithmus schnell oder korrekt sein muss auf *allen* Eingaben. In manchen Anwendungen ist es möglich, dass die Art der Eingabe sehr beschränkt ist und dass ein effizienter Algorithmus für solche Teilprobleme existiert. Das Hamiltonkreisproblem ist beispielsweise NP-schwer, aber es kann gezeigt werden, dass es einen Algorithmus gibt, der einen Hamiltonkreis findet in "fast jedem" Graphen der einen hat. Solche Resultate wurden oft erreicht mit Wahrscheinlichkeitsverteilungen über die Eingabe und zeigen dann, dass gewisse Heuristiken das Problem mit grosser Wahrscheinlichkeit lösen.
- **Approximations-Algorithmen:** Wenn man schwere Probleme nur schwer exakt lösen kann, muss man sich mit Näherungslösungen zufrieden geben, die man schnell berechnen kann.

Mit Approximations-Algorithmen suchen wir somit möglichst gute zulässige Lösungen, die schnell berechnet werden können.

Betrachte ein (Minimieerungs)Optimierungsproblem (Die Definition für ein Maximierungsproblem ist analog). Es sei $C_{\mathrm{OPT}}(\sigma)$ der Wert der optimalen Lösung auf der Eingabe σ und es sei $C_A(\sigma)$ der Wert eines Algorithmus A auf derselben Eingabe σ. Dann heisst A ein $\alpha(n)$-*Approximations-Algorithmus* falls gilt, dass

$$\frac{C_A(\sigma)}{C_{\mathrm{OPT}}(\sigma)} \leqslant \alpha(n)$$

für alle Eingaben σ der Länge n. Der Wert $\alpha(n)$ heisst dabei *Approximations Faktor* oder auch *relative Güte*.

Ein 1-Approximations-Algorithmus berechnet z.b. immer die exakte Lösung. Beachte, dass α nicht konstant sein muss, und wir z.B. von einer $\log n$-Approximation sprechen können.

Beispiel: Load Balancing

Betrachten wir das *Load Balancing Problem* (auch *Lastenverteilungsproblem* genannt) das zum Beispiel unter anderem auftaucht wenn mehrere Server gemeinsam eine Liste von Anfragen (oder *requests*) bearbeiten. Zur Illustration nehmen wir an, dass alle Machinen (Server) gleich sind und definieren das Problem folgendermassen: Wir haben m Maschinen $M_1, \ldots, M_m$ und eine Liste von n Anfragen oder Aufgaben; jede Aufgabe (oder *job*, *task*) j hat eine bestimmte Bearbeitungsdauer t_j. Wir möchten die Aufgaben möglichst gut auf die Maschinen aufteilen.

Für jede Aufteilung sei $A(i)$ die Menge der Aufgaben, die der Maschine M_i zugeordnet sind. Dann ist diese Maschine für eine totale Zeit von

$$T_i = \sum_{j \in A(i)} t_j$$

ausgelastet, und wir nennen dies die *load* (oder *Last*) der Maschine M_i. Wir möchten den *makespan* minimieren, den wir als die maximale Last einer Maschine definieren, also $T = \max_i T_i$.

Stop and Think Das Problem eine Verteilung zu finden, die den makespan minimiert ist NP-schwer.

Betrachten wir einen einfachen Greedy-Algorithmus der die Aufgaben zuerst absteigend nach Bearbeitungsdauer sortiert und danach nur einen Durchlauf durch die sortierte Aufgabenliste macht: wenn er Aufgabe j zu bearbeiten hat, dann wird j derjenigen Maschine zugeordnet, deren Last soweit am geringsten ist.

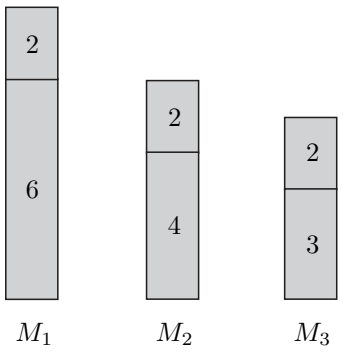


Abb.: Der Greedy-Algorithmus angewendet auf drei Maschinen und Aufgaben der Grössen 2, 3, 4, 6, 2, 2.

SORTED-LOADBALANCE(m, t)

```
   // Sortiere Aufgaben absteigend.
1  SORT(t)
2  for i = 1 to m
3       A[i] = ∅
4       T[i] = 0
5  for j = 1 to n
6       k = arg min_k T[k] // Maschine mit geringster Auslastung.
        // Ordne Aufgabe j der Maschine k zu
7       A[k] = A[k] ∪ {j}
8       T[k] = T[k] + t[j]
```

Wir sehen, dass wenn mir weniger als m (die Anzahl der Maschinen) Aufgaben haben, der Algorithmus optimal ist, da der optimale makespan mindestens $\max_j t_j$ ist (think about it) und wir jede Aufgabe einer eigenen Maschine zuordnen. Ansonsten überlegen wir uns folgendes: Es sei T der makespan der resultierenden Aufteilung. Wir möchten zeigen, dass T nicht zu viel grösser als T^* ist - der minimal mögliche optimale makespan. Um dies zu zeigen, möchten wir unsere Lösung mit T^* vergleichen, kennen dies aber nicht und können es auch nicht berechnen. Für die Analyse benötigen wir deshalb eine *lower bound* (untere Schranke) auf das Optimum - die uns garantieren wird, dass die optimale Lösung nicht kleiner ist.

Es gibt nun mehrere Kandidaten. Eine Möglichkeit ist es zu sehen, dass mit m Maschinen eine mindestens einen $\frac{1}{m}$ Anteil des Gesamtaufwandes aufbringen muss, also

Beobachtung 1 *Der optimale makespan ist mindestens*

$$T^* \geqslant \frac{1}{m} \sum_j t_j$$

Unglücklicherweise ist diese Abschätzung alleine zu schwach um uns zu helfen: Nehme an, dass es einen Job/Aufgabe gibt, die viel länger braucht als alle anderen. In einer genug extremen Version davon, wird die optimale Lösung sein, diese Aufgabe auf einer eigenen Maschine zu bearbeiten. In diesem Fall ist der Greedy-Algorithmus optimal, aber Beobachtung 1 ist zu schwach um dies zu zeigen (think about it).

Nützlich ist zusätzlich folgende Beobachtung.

Beobachtung 2 *Der optimale makespan ist mindestens*

$$T^* \geqslant \max_j t_j$$

Wir können uns (mit dem Schubfachprinzip) auch noch folgendes überlegen

Beobachtung 3 *Gibt es mehr als m Aufgaben, so ist $T^* \geqslant 2t_{m+1}$ - für t_{m+1} in absteigend sortierter Anordnung.*

Beweis. Betrachte die ersten $m+1$ Aufgaben in der absteigend sortierten Anordnung. Jede davon braucht mindestens eine Bearbeitungsdauer von t_{m+1}. Es gibt $m+1$ Aufgaben davon und nur m Maschinen, deshalb muss mindestens eine davon zwei von diesen Aufgaben bearbeiten. Diese Maschine hat deshalb eine Bearbeitungszeit von mindestens $2t_{m+1}$.

Damit können wir nun zeigen:

Beobachtung 4 *Der Algorithmus* Sorted-LoadBalance *erzeugt eine Aufgabenverteilung mit makespan $T \leqslant \frac{3}{2}T^*$.*

Beweis. Betrachte die Maschine M_i die die grösste Auslastung unter unserer Aufgabenverteilung T hat. Wenn M_i nur eine Aufgabe hat, so ist unsere Aufteilung optimal (Beobachtung 2).

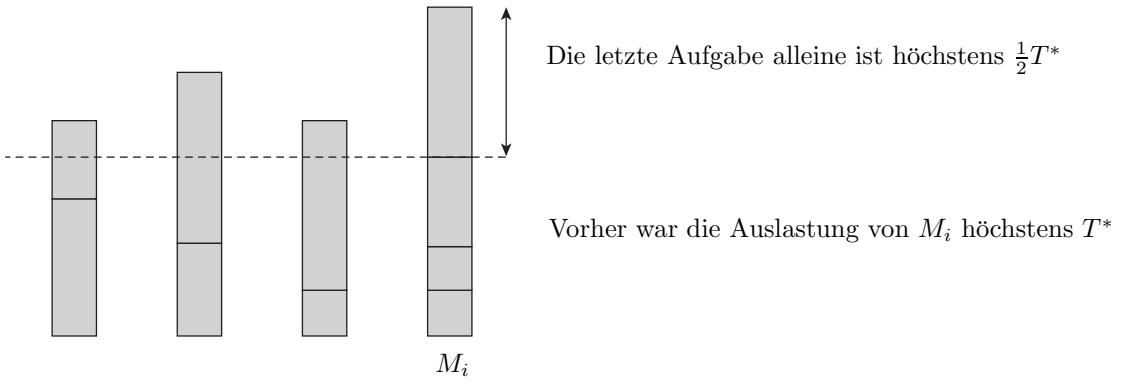


Abb.: Beweisskizze

Ansonsten nehmen wir an, dass M_i mindestens zwei Aufgaben zu bearbeiten hat, und es sei t_j die letzte Aufgabe, die dieser Maschine zugeordnet wurde. Wir wissen, dass $j \geqslant m+1$, da der Algorithmus die ersten m Aufgaben auf die m verschiedenen Maschinen verteilt. Daher ist $t_j \leqslant t_{m+1} \leqslant \frac{1}{2}T^*$ mit Beobachtung 3.

Nun als j der Maschine M_i zugeordnet wurde, hatte M_i die geringste Auslastung. Diese war zu diesem Zeitpunkt $T_i - t_j$ und daher hat jede andere Maschine mindestens diese Auslastung auch am Ende, und daher $\sum_k T_k \geqslant m(T_i - t_j)$ oder anders ausgedrückt,

$$T_i - t_j \leqslant \frac{1}{m}\sum_k T_k.$$

Aber $\sum_k T_k$ ist gerade totale Last aller Aufgaben $\sum_j t_j$ da jede Aufgabe genau einer Maschine zugeordnet wird, daher wissen wir mit Beobachtung 1, dass

$$T_i - t_j \leqslant T^*.$$

Damit ist (mit Beobachtung 3)

$$T_i = (T_i - t_j) + t_j \leqslant T^* + t_j \leqslant \frac{3}{2}T^*.$$

Christofides' Algorithmus

Ein weiteres Beispiel ist ein Algorithmus, der das metrische *Travelling Salesman Problem* mit relativer Güte $\frac{3}{2}$ approximiert. Er heisst nach seinem Erfinder *Christofides' Algorithmus* (vgl. Vorlesung).

Das *Travelling Salesman Problem* (TSP) ist die Berechnung eines kürzesten Hamilton Zyklus in einem gewichteten Graphen. Das allgemeine TSP kann nicht approximiert werden, der metrische Spezialfall allerdings schon. Dieser Speziallfall verlangt, dass die Kantengewichte die *Dreiecksungleichung* erfüllen:

$$\forall u, v, w : l(u, w) \leqslant l(u, v) + l(v, w)$$

Dies ist gegeben für geometrische Graphen, bei denen die Knoten Punkte in der Ebene (oder in einem höher-dimensionalen Raum) sind, Kanten direkte Verbindungen, und die Längen der üblichen euklidschen Metrik entsprechen. Nehmen wir hierfür an (ohne Verlust der Allgemeinheit), dass der gegebene Graph vollständig ist (K_n).

Stop and Think: Zeige dass folgender Algorithmus eine Approximation mit relativer Güte 2 berechnet:

TSP-Approx-2(G)
1 $T = \text{MST}(G)$ // $\mathcal{O}(n^2 \log n)$
2 Nummeriere Knoten mit Preorder-DFS-Traversierung von T
3 Besuche Knoten in dieser Nummerierung (überspringe Knoten die schon besucht wurden)

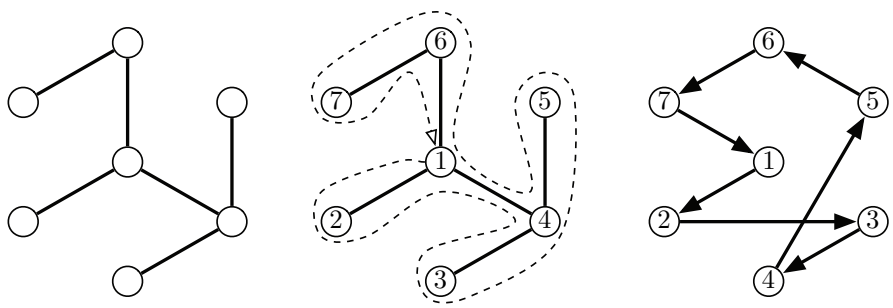


Abb.: Ein Minimaler Spannbaum T, DFS Traversierung von T, Approximierte Lösung

Obiger Algorithmus kann mit Christofides' Algorithmus zu einer relativen Güte von $\frac{3}{2}$ verbessert werden. Wie oben wird zuerst ein minimaler Spannbaum berechnet. Zusätzlich berechnen wir ein leichtestes Matching und eine Euler Tour folgendermassen:

TSP-Christofides(G)

1 $T = \text{MST}(G)$ // $\mathcal{O}(n^2 \log n)$
2 $S = \{v \in T : \deg_T(v)$ ist ungerade$\}$ // Es gilt: $|S|$ ist gerade.
3 Berechne auf dem durch S induzierten Teilgraphen ein leichtestes Matching M
4 Berechne eine Euler-Tour $E = (u_1, u_2, \ldots)$ auf $T \cup M$
// Alle Knoten in $T \cup M$ haben geraden Grad
5 Entferne in E Wiederholungen von Knoten, so dass man E' enthählt
6 **return** E'

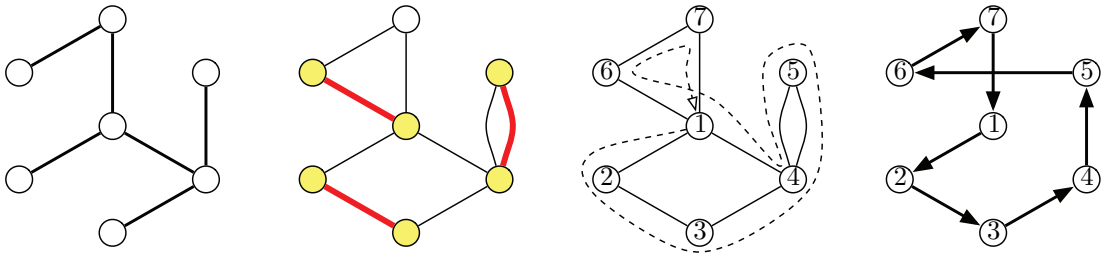


Abb.: Ein MST T, leichtestes Matching M auf S, Euler Tour E, Approximierte Lösung

Online Algorithmen

Mit Approximationsalgorithmen versuchen wir optimale Lösungen zu approximieren für z.b. NP-schwere Probleme. Mit Online Algorithmen versuchen wir möglichst gute Lösungen zu finden für Probleme die nicht notwendigerweise schwer rechnerisch/komplex im NP Sinne sind, aber bei denen ein Algorithmus nicht alle *Informationen* zu Beginn hat.

Wir sind mit einem *Online* Problem konfrontiert, wenn nicht die gesamte Eingabe zu Beginn bekannt ist, aber stückweise präsentiert wird und der Algorithmus die Eingabe verarbeiten muss wann sie ankommt. Vergleiche dies zum Beispiel mit dem klassischen Sortierproblem, bei dem die gesamte Eingabe zu Beginn bekannt ist. Viele Datenstrukturen sind Beispiele solcher Online Algorithmen - so müssen z.b. Wörterbuch Datenstrukturen wie ein balancierter Baum, Einfüge und Abfrage Operationen schnell unterstützen, ohne zu wissen, welche Abfragen in der Zukunft gemacht werden.

Da wir in einem solchen Fall nicht die restliche Eingabe kennen, kann es sein, dass wir nicht optimale Entscheidungen machen können. Ob eine Entscheidung gut oder schlecht ist hängt von zukünftigen und damit unbekannten Ereignissen ab. Dies führt natürlicherweise zu einer Einbusse in der Qualität der Lösung - der inherente Preis, die Zukunft nicht zu kennen. Dieser Einbuss-Faktor nennen wir den *Competitive Ratio* eines Online-Algorithmus. Ein Online-Algorithmus A heisst dabei α-*kompetitiv*, falls für alle Eingaben σ gilt

$$C_A(\sigma) \leqslant \alpha \cdot C_{\text{OPT}}(\sigma) + \delta$$

wobei C_A und C_{OPT} die Kosten der Lösung von A bzw. des Optimums definiert und δ eine Konstante ist.

Beachte insbesondere, dass dies über alle Eingaben und damit auch für den Worst-Case gelten muss, dies da Online Algorithmen zu robusten Lösungen kommen sollten, die was auch passiert nicht zu weit weg von der Lösungsqualität eines allwissenden Algorithmus sind.

Online Algorithmen treten in verschiedenen Bereichen auf, wie z.B. Network Routing, Cache Paging, Scheduling etc.

Das Rent-Or-Buy Problem

Hier ist ein eifaches veranschaulichendes Beispiel, das *rent-or-buy* Problem. Sagen wir, dass wir Ski fahren gehen wollten. Wir können entweder ein Paar Ski mieten für 50 € oder sie kaufen für 500 €. Aber wir wissen noch nicht, ob wir überhaupt gerne Ski fahren, also mieten wir sie für den ersten Tag. Und dann gehen wir nochmals am nächsten Tag, und am übernächsten Tag und schon bald merken wir, dass wir die Ski besser gleich zu Beginn gekauft hätten. Die optimale Strategie wäre gewesen: fahren wir an mehr als 10 Tagen Ski, dann hätten wir sie kaufen sollen,

ansonsten mieten. Was wenn wir dies nicht zu Beginn wüssten (um das Problem ein Online Problem zu machen)?

Stop and Think: Überlege, dass die Strategie niemals zu kaufen und immer nur zu mieten, beliebig schlecht ist.

Nennen wir folgende Strategie BETTER-LATE-THAN-NEVER: Wir mieten die Ski solange bis wir merken, dass wir sie hätten zu Beginn kaufen sollen, dann kaufen wir sie. In obigem Beispiel: miete 9 Tage lang, dann kaufe die Ski. Sind die Mietkosten m und der Kaufpreis p, dann mieten wir zuerst $\lceil \frac{p}{m} \rceil - 1$ Mal und dann kaufen wir.

Beobachtung 5 *Der Algorithmus* BETTER-LATE-THAN-NEVER *hat einen competitive ratio* $\leqslant 2$. *Falls der Kaufpreis* p *ein Vielfaches der Mietkosten* m *ist, dann ist er* $2 - \frac{m}{p}$ *kompetitiv.*

Beweis. Betrachten wir zwei Fälle. Falls wir an weniger als $\lceil \frac{p}{m} \rceil$ Tagen Ski fahren (z.B. 9 oder weniger Tage für den Fall dass $p = 500$ und $m = 50$), dann ist der Algorithmus optimal. Der Algorithmus hat nie gekauft und auch das Optimum tut nicht.

Falls wir an mehr als $\lceil \frac{p}{m} \rceil$ Tagen Ski fahren, dann ist die optimale Lösung die Ski am Anfang zu kaufen, also $C_{\text{OPT}} = p$. Der Algorithmus BETTER-LATE-THAN-NEVER hat $\lceil \frac{p}{m} \rceil - 1$ gemietet und dann gekauft, also sind die Kosten $m\left(\lceil \frac{p}{m} \rceil - 1\right) + p$. Dies ist immer weniger als $2p$ und ist gleich $2p - m$ wenn p ein Vielfaches von m ist.

Der Worst-Case ist der zweite Fall mit $\frac{2p}{p} = 2$ (bzw. $\frac{2p-m}{p} = 2 - \frac{m}{p}$ falls p ein Vielfaches ist von m) und gibt den behaupteten α Wert.

Beobachtung 6 *Der Algorithmus* BETTER-LATE-THAN-NEVER *hat den besten competitive ratio für das rent-or-buy Problem für deterministische Algorithmen wenn* p *ein Vielfaches von* m *ist.*

Beweis. Stop and think! Betrachte den Fall, dass der Tag an dem die Skis gekauft werden, der letzte Tag ist, an dem Ski gefahren wird.

Das Lift-Problem

Man steht vor einem Lift/Aufzug und drückt den Knopf. Aber wie lange wird es dauern, bis der Lift kommt, falls überhaupt? Wie lange sollte man warten, bis man aufgibt und die Treppe benutzt?

Sagen wir, es dauert eine Zeit L um mit dem Lift (wenn er da ist) von unserem Stockwerk zu unserem Zielstockwerk zu kommen. Die Treppe zu nehmen braucht Zeit T, z.B. könnte $L = 15$ und $T = 45$ Sekunden sein.

Wie lange sollte man warten? Welche Strategie hat den besten *competitive ratio*? Die Antwort hier ist warte 30 Sekunden, dann nehme die Treppe - im Allgemeinen sollte man $T - L$ Sekunden warten. Dies ist genau die *better-late-than-never* Strategie, da wir die Treppe nehmen sobald wir realisieren, dass wir sie zu Beginn hätten nehmen sollten. Falls der Lift vor 30 Sekunden $(T - L)$ kommt, ist unsere Strategie optimal, ansonsten ist OPT = 45. Wir brauchen $30 + 45$ Sekunden (im Allgemeinen $T - L + T$), und damit sind wir bei $(30 + 45)/45 = \frac{5}{3}$, oder im Allgemeinen $2 - \frac{L}{T}$.

Dies ist eigentlich wieder das *Rent-Or-Buy* Problem von vorher. Andere ähnliche Probleme: Wann ist es wert Code zu optimieren, wann soll die Harddisk anhalten zwischen Zugriffen, und viele andere.