Praktische Informatik I – Der Imperative Kern
Der Nutzen von Heuristiken

Prof. Dr. Stefan Edelkamp

Institut für Künstliche Intelligenz
Technologie-Zentrum für Informatik und Informationstechnik (TZI)
Am Fallturm 1, 28359 Bremen
+49-(0)421-218-64007
www.tzi.de/~edelkamp

Transparenz

In diesem Abschnitt lernen wir effiziente Tiefensuch-Varianten kennen, die durch den Einsatz von unteren Schranken deutlich schneller werden.

Der Begriff Heuristik wird Archmedis Spruch Heureka bei der Entdeckung des Auftriebsprinzips zugesprochen. In der Informatik sind Heuristiken Daumenregeln, die zur Vereinfachung eines Algorithmus angewendet werden.

Wir werden darunter im Folgenden richtungsweisende Hilfestellungen in Form von unteren Schranken, die den Suchprozess beschleunigen.

Outline

Bitte eine gerechte Teilung

Betrachten wir nun folgendes Zahlenaufteilungsproblem

Gegeben Eine Menge $N = \{0, \ldots, n-1\}$ von Objekten mit Größe $a_0, \ldots, a_{n-1}$
Gesucht Gleichmäßige Aufteilung von N in $S \subseteq N$ und $N \setminus S$ mit $\sum_{i \in S} a_i = \sum_{j \in N \setminus S} a_j$.

Das Problem ganzzahlig nicht lösbar falls $\sum_{i \in N} a_i$ ungerade ist.

Das Zahlenaufteilungsproblem ist Basis für viele verwandte Packungsprobleme - Es ist nicht bekannt, ob man es effizient lösen kann.

Die gierige (engl. greedy) Heuristik in Programm 2 legt das nächste Objekt auf die jeweils kleiner Seite und berechnet die entstehende neue Differenz der beiden Seiten.

Dabei erweist es sich als nützlich, wenn die Elemente absteigend sortiert sind.

Programm 1: Zahlenaufteilung: Gierige und vollständige Lösung.

```
import java.util.Arrays;
import java.util.Random;
public class Partition
{
    int [] a;
    int max;
    public Partition(int n) {
      max = Integer.MAX_VALUE;
      Random r = new Random();
      a = new int[n];
      for(int i=0;i<n;i++) a[i] = r.nextInt(20);
      Arrays.sort(a); // increasing -> decreasing
      for(int i=0,j=n-1; i < (n/2); i++, j--) { int t = a[i]; a[i] = a[j];a[j] = t; }
      for (int i=0; i<n; i++) System.out.print(a[i]+",");
      completeGreedy(0,0);
      System.out.println("remaining difference is "+max);
    }
    public int greedy() {
      int diff = 0;
      for (int i=0;i<a.length-1;i++)
        if (diff < a[i]) diff = a[i] - diff; else diff = diff - a[i];
      return diff;
    }
```

Optimale Lösung

Die Lösung wird zu einer vollständige Suche erweitert, die im Backtrackschritt das ablegen des Objekt auch auf die größere Seite erlaubt.

Dabei wird letztendlich die bestmögliche Aufteilung berechnet.

Programm 2: Zahlenaufteilung: Gierige und vollständige Lösung.

```
    public void completeGreedy(int i, int diff) {
      System.out.println(i+" ["+diff+"]");
      if (i == a.length) { if (diff < max) max = diff; return; }
      completeGreedy(i+1,diff < a[i] ? (a[i] - diff) : (diff - a[i]));
      completeGreedy(i+1,diff+a[i]);
    }
}
```

Outline

Schieb mich

Das 15-Puzzle gehört zu den beliebtesten Spielen in den Kinderzimmern. Duch geeignetes Bewegen der Spielsteine muss die Anordung 0,...,15 hergestellt werden.

Untere Schranke Manhatten-Distanz Summe der horizontalen und vertikalen Abstände der Einzelspielsteine von ihrer Zielposition.

Ein Tiefensuch-Löser ist aufgrund seiner schnellen Knotengenerierung und die Richtungsweisung in der Heuristik in dem Problem sehr effektiv, obwohl es viele Zustände gibt, die auf verschiedenen Pfaden erreicht werden können.

Programm 3 zeigt den Konstruktor und die Instanzvariablen

Um die Operatorenauswahl zu beschleunigen , werden, abhängig von der Position des Leerfeldes (engl. blank), die Anzahl der Operatoren und die jeweiligen Nachfolgepositionen bestimmt.

Zur inkrementellen Berechnung der Heuristik wird zudem ein drei-dimensionales Array **inc** vorberechnet.

Programm 3: 15-Puzzle-Löser: Konstruktor und Zufallsstellung.

```
import java.util.Random;
public class FifteenPuzzle
{
    public static final int X = 4;
    public static final int SIZE = 16;
    int [] s;
    public class Operators {
        int num; /* number of applicable oprs: 2..4 */
        int [] pos = new int[4];
    }
    Operators [] oprs = new Operators[SIZE];
    int [][][] inc = new int [SIZE][SIZE][SIZE]; /* incr eval func: tile, source, dest */
    int thresh; /* search cutoff threshold */
    long generated; /* number of states generated per iteration */
    long total; /* total number of states generated */
    public FifteenPuzzle() {
      s = new int[SIZE];
      for (int i=0;i<SIZE;i++) oprs[i] = new Operators();
      for (int blank = 0; blank < SIZE; blank++) { /* for each possible blank position */
        oprs[blank].num = 0; /* no moves initially */
        if (blank > X - 1) oprs[blank].pos[oprs[blank].num++] = blank - X; /* add move up */
        if (blank % X > 0) oprs[blank].pos[oprs[blank].num++] = blank - 1; /* add move left */
        if (blank % X < X - 1) oprs[blank].pos[oprs[blank].num++] = blank + 1; /* add move right */
        if (blank < SIZE - X) oprs[blank].pos[oprs[blank].num++] = blank + X; /* add move down */
      }
      for (int tile = 1; tile < SIZE; tile++) /* all physical tiles */
        for (int source = 0; source < SIZE; source++) /* all legal source positions */
          for (int destindex = 0; destindex < oprs[source].num; destindex++) {
            int dest = oprs[source].pos[destindex]; /* dest is new blank position */
            inc[tile][source][dest] =
              Math.abs((tile % X) - (dest % X)) - Math.abs((tile % X) - (source % X)) +
              Math.abs((tile / X) - (dest / X)) - Math.abs((tile / X) - (source / X));
          }
    }
```

Zufallsstellungen und Iterierte Tiefensuche

Das Erzeugen einer Zufallsstellung wird dadurch geschickt gelöst, dass jeweils zwei zufällig gewählte Plättchen getauscht werden.

Dabei wird durch eine gerade Anzahl von Tauschoperationen gewährleistet, dass keine unlösbare Stellung erzeugt wird.

Tatsächlich sind nur die Hälfte der 16! möglichen Anordnungen lösbar.

Die iterierte Tiefensuche in Programm 3 zeigt die rekursive Suchprozedur und die übergeordnete Schleife, die meherer Problemstellungen auswürfelt und löst.

Die Statistik über die optimale Suchtiefe und die Anzahl besuchter Zustände wird im Hauptprogramm ausgegeben.

Alle Berechnungen an einem Zustand sind in konstanter Zeit durchzuführen.

Programm 4: 15-Puzzle-Löser: Iterierte Tiefensuche.

```
  public int search (int blank, int oldblank,int g,int h) {
    int newblank; /* blank position in new state */
    for (int index = 0; index < oprs[blank].num; index++) /* for each appl opr */
      if ((newblank = oprs[blank].pos[index]) != oldblank){ /*not inv last move */
        int tile = s[newblank]; /* tile moved is in new blank position */
        int newh = h + inc[tile][newblank][blank]; /* new heuristic est */
        generated++; /* count nodes generated */
        if (newh+g+1 <= thresh) { /* less than search cutoff */
          s[blank] = tile; s[newblank] = 0;
          if ((newh == 0) || (search(newblank, blank, g+1, newh) == 1)) return 1; /* exit with success */
          s[newblank] = tile; s[blank] = 0; /* undo move before doing next */
        }
    }
    return 0; /* exit with failure */
  }
  public void driver() {
    for (int problem = 1; problem <= 5; problem++) { /* for each initial state */
      int blank = generate(); /* input initial state */
      int initeval = 0;
      for (int pos = 0; pos < SIZE; pos++)
        if (s[pos] != 0) /* blank isn't counted in Manhattan distance */
          initeval += Math.abs((pos % X) - (s[pos] % X)) + Math.abs((pos / X) - (s[pos] / X));
      thresh = initeval; /* initial threshold is initial h */
      total = 0; /* initialize total nodes generated */
      int success = 0;
      do { /* depth-first iterations until solution is found */
        generated = 0; /* initialize number generated per iteration */
        success = search(blank, -1, 0, initeval); /* perform search */
        total = total + generated; /* keep track of total nodes per problem */
        thresh += 2; /* threshold always increases by two */
        System.out.println("..... Problem "+ problem +" cost "+ (thresh-2) + " total nodes generated "+ total);
      } while (success == 0); /* until solution is found */
      System.out.println("Solved Problem "+ problem +" cost "+ (thresh-2) + " total nodes generated "+ total);
    }
  }
```

Outline

Perfektes Hashing

Ranking ist die eindeutige Abbildung einer Permutation auf eine Zahl in $[0, \ldots, n! - 1]$. Unranking erzeugt, gegeben die Zahl, die Permutation.

Eine schnelle Ranking- und Unranking-Funktion wurde von Myrvold und Ruskey basierend auf folgender Erkenntnis konstruiert.

> *Eine zufällige Permutation kann dadurch erzeugt werden, dass für alle k aus $n - 1, \ldots, 1$ die Werte von $\pi[k]$ mit $\pi[rand(k)]$ getauscht werden, wobei $rand(k)$ eine Zufallszahl im Bereich $0, \ldots, k$ ist.*
>
> *Die entscheidende Beobachtung ist, dass alle so erzeugten Permutationen die gleiche Wahrscheinlichkeit haben also uniform verteilt sind und jede Sequenz von Zufallszahlen genau einer Zahl in $[0, \ldots, n! - 1]$ entspricht.*

Implementierung

Die Prozedur 5 realisiert diese Berechnungen und stellt eine entsprechende Testroutine zur Verfügung.

Sie brauchen offensichtlich jeweils eine Rechenzeit, die proportional zu n – also linear in der Eingabegröße – ist.

Vorsicht Die ermittelten Rangzahlen der Permutationen entsprechen nicht der natürlichen oder lexikographischen Ordnung

Anwendung Die Anwendung zur Lösung von Permutationsspielen wie dem 15-Puzzle ist unmittelbar

Anstatt zur Duplikatsvermeidung ganze Zustände zu speichern, reicht es, ein Bit in einem Bitvektor der Länge $[0, \ldots, n! - 1]$ zu setzen.

Desweiteren kann der Zustand selbst aus der Adresse rekonstruiert werden, was eine speicherplatzfreundliche Exploration auf einen Bitvektor ermöglicht.

Programm 5: Ranking und Unranking in Linearzeit.

```
public class Rank
{
  static final int N = 9;
  int [] pi = new int[N];
  int [] inversepi = new int[N];
  private void unrank(int n, int r, int pi[]) {
    if (n>0) {
      int tmp = pi[n−1]; pi[n−1] = pi[r%n]; pi[r%n] = tmp;
      unrank(n−1,r/n,pi);
    }
  }
  private int rank(int n, int pi[], int inversepi[]) {
    if (n==1) return 0;
    int s = pi[n−1];
    int tmp = pi[n−1]; pi[n−1] = pi[inversepi[n−1]]; pi[inversepi[n−1]] = tmp;
    tmp = inversepi[s]; inversepi[s] = inversepi[n−1]; inversepi[n−1] = tmp;
    return s + n∗rank(n−1,pi,inversepi);
  }
  public void test(int value) {
    pi[0] = 5; pi[1] = 8; pi[2] = 3; pi[3] = 6; pi[4] = 1; pi[5] = 0; pi[6] = 2; pi[7] = 4; pi[8] = 7;
    for (int i=0;i<N;i++) inversepi[pi[i]] = i;
    System.out.println("rank value of permutation is "+rank(N,pi,inversepi));
    for (int i=0;i<N;i++) pi[i] = i;
    unrank(N,value,pi);
    for (int i=0;i<N;i++) System.out.print(pi[i]+", "); System.out.println();
  }
```

Outline

Bitte in die Mitte

Das Solitärspiel (engl. peg solitaire) ist ein bekanntes und herausfordendes Einpersonenspiel. Hier geht es darum durch iteriertes Springen und begleitenes Wegnehmen von Steinen, die Anzahl auf 1 zu verringern. Es ist sogar möglich, dass der letzte Stein in der Mitte des Brettes liegen bleibt.

Die vollständige Lösung des (Englischen bzw. Europäischen) Spiels auf dem Computer durch Backtracking benötigt etwas Zeit, so dass wir uns in Programm 7 auf eine einfachere Eingabe beschränken.

Heuristiken beschleunigen den Lösungsprozess und man kann alle Spielstellungen mit geeigneten Datenstrukturen zählen

Auch für das Solitärspiel lassen sich effiziente Ranking- und Unranking-Funktionen jeweils ein Paar für jede Tiefe definieren. Dabei nutzt man Binomialkoeffizienten

Programm 6: Die Lösung eines Solitär-Spiels.

```
public class Peg
{
    final int PEGS = 25;
    final int XDIM = 7;
    final int YDIM = 6;
    private int [] solx = new int [PEGS];
    private int [] soly = new int [PEGS];
    private char B[][] = {
        {'—','—','x','x','—','—'},
        {'—','—','x','x','—','—'},
        {'x','x','x','x','x','x'},
        {'x','x','o','x','x','x'},
        {'x','x','x','x','x','x'},
        {'—','—','x','x','—','—'},
        {'—','—','x','x','—','—'} };
    /**
     * Constructor for objects of class Peg
     */
    public Peg() {
        solve(PEGS);
    }
```

Programm 7: Die Lösung eines Solitär-Spiels.

```
public void solve(int pegs) {
  if (pegs == 1) {
    for (int i=PEGS-1;i>0;i--)
      System.out.print("("+solx[i]+","+soly[i]+")");
    System.exit(1);
  }
  for (int i=0;i<XDIM;i++)
    for (int j=2;j<YDIM;j++)
      if (B[i][j] == 'o' && B[i][j-1] == 'x' && B[i][j-2] == 'x') {
        B[i][j-2] = B[i][j-1] = 'o'; B[i][j] = 'x';
        solx[pegs-1] = i; soly[pegs-1] = j;
        solve (pegs-1);
        B[i][j-2] = B[i][j-1] = 'x'; B[i][j] = 'o';
      }
  for (int i=0;i<XDIM;i++)
    for (int j=0;j<YDIM-2;j++)
      if (B[i][j] == 'o' && B[i][j+1] == 'x' && B[i][j+2] == 'x') {
        B[i][j+2] = B[i][j+1] = 'o'; B[i][j] = 'x';
        solx[pegs-1] = i; soly[pegs-1] = j;
        solve (pegs-1);
        B[i][j+2] = B[i][j+1] = 'x'; B[i][j] = 'o';
      }
  for (int i=2;i<XDIM;i++)
    for (int j=0;j<YDIM;j++)
```

Outline

Logistik für Anfänger

Ein Handlungsreisender möchte in einer Rundreise n Städte nacheinander genau einmal besuchen und dabei die kürzeste Strecke zurücklegen.

Für eine Distanzmatrix $d_{i,j}$ mit $0 \leq i, j \leq n-1$ eine Permutation π der Zahlen $0, \ldots, n-1$ gesucht, für die $d_{\pi_{n-1},\pi_0} + \sum_{i=1}^{n-1} d_{\pi_{i-1},\pi_i}$ minimal ist.

Das Problem taucht als Teilproblem in der Logistik auf, z.B. bei der Auslieferungen von Gütern (z.B. aus einem Depot). Die Distanzen $d_{i,j}$ werden dabei durch ein Verfahren zur Bestimmung der kürzesten Wege (z.B. mit dem Algorithmus von Dijkstra) bestimmt.

Das Handlungsreisendenproblem ist eine Optimierungsaufgabe, für den man keinen effizienten Algorithmus kennt.

Dennoch ist das Handlungsreisendenproblem in der Praxis für moderate Städteanzahlen gut zu lösen. Dies liegt vor allem an dem Einsatz von unteren Schranken – sogenannten Heuristiken .

Depth-First Branch-and-Bound

Diese Heuristiken werdem in dem Branch-and-Bound Ansatz dazu genutzt, Teillösungen auszuschließen, die nicht mehr zur Verbesserung der aktuell besten Lösung führen können.

Anders als bei der iterierten Tiefensuch-Lösung wird nicht beim Erreichen der ersten Lösung gestoppt, sondern solange weitergesucht, bis keine Verbesserung der Lösung möglich ist.

Programm 8 zeigt die Teilklassen die einen dynamischen Speicherverbrauch vermeiden. Desweiteren wird der Konstruktor der Klasse TSP angegeben bei dem die Nachfolger nach Distanzen sortiert werden, um gute Lösungen möglichst schnell zu finden.

Besonderheit Der Vektor der besuchten Städte wird durch eine einzige Zahl realisiert, die als Bitvektor interpertiert wird. Damit kann jeder Nachfolger in konstanter Zeit erzeugt werden, jedoch ist die Anzahl der Städte so auf die Wortbreite des Computers beeschränkt.

Programm 8: Problem des Handlungsreisenden: Initialisierungen.

```
public class TSP
{
    public class State {
      int h, g, depth, city;
      long used;
      public State() { used = 0L; g = h = depth = city = 0; }
    }
    public class Memory {
      int top, max;
      State [] old;
      public Memory(int n) {
        max = n*n; top = 0; old = new State[max+1];
        for (int i=0;i<max+1;i++) old[i] = new State();
      }
    }
    static final int N = 50;
    public TSP() {
      start = 0;
      Random r = new Random(100);
      for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
          dist[i][j] = (i == j) ? 0 : r.nextInt(100);
      for (int i = 0; i < N; i++) {
        newState[i] = new State();
        for (int j = 0; j < N; j++)
          far[i][j] = j;
        for (int j = 0; j < N-1; j++)
          for (int k = j+1; k < N; k++)
            if (dist[i][far[i][j]] < dist[i][far[i][k]]) {
              int temp = far[i][j]; far[i][j] = far[i][k]; far[i][k] = temp;
            }
      }
```

Implementierung

Das Programm 10 gibt die eigentliche Branch-and-Bound Suche wider: Wird eine bessere Lösung als die bis dato best-bekannte gefunden, so wird sie ausgegeben.

In der Variable α wird die aktuell beste Lösung vorgehalten.

Hier sehen wir, wie die Bits der Zahl **used** durch Bitoperationen mit dem Logischen Oder (|) gesetzt und mit dem Logischen Und (&) gelöscht werden. Durch den Logischen Negationsoperor $\sim$ werden alle Bits in einem Vektor umgedreht.

Verschiebungen können sowohl nach rechts ($>>$) oder nach links erfolgen ($<<$), wobei Nullen nachgeschoben werden. Numerisch interpretiert entprechen die Operation einer Division oder einer Multiplikation mit einer Zweierpotenz

Programm 9: Problem des Handlungsreisenden: Lösungssuche.

```
public int solve(int h) {
  expansions = 0;
  int alpha = Integer.MAX_VALUE;
  int top = stack.top++;
  stack.old[top].g = stack.old[top].depth = 0;
  stack.old[top].city = start;
  used = 0L;
  stack.old[top].h = heuristic(h,0,0,0);
  System.out.println("Heuristic value at root = " + stack.old[top].h);
  stack.old[top].used = used = (1L << start);
  while (stack.top != 0) {
    top = --stack.top;
    int depth = stack.old[top].depth;
    int city = stack.old[top].city;
    tour[depth] = city;
    if (depth == N - 1) {
      if (stack.old[top].g + dist[city][start] < alpha) {
        alpha = stack.old[top].g + dist[city][start];
        System.out.println(" cost: " + alpha +" (" + expansions + ")");
      }
      continue;
    }
    used = stack.old[top].used;
    int cost = stack.old[top].g;
    int opindex = 0;
```

Programm 10: Problem des Handlungsreisenden: Lösungssuche.

```
      for(int i=0; i < N; i++) {
        if (((used >> far[city][i]) & 1L) > 0) continue;
        int newcity = far[city][i];
        newState[opindex].depth = depth+1;
        next[opindex] = opindex + 1;
        int g = cost + dist[city][newcity];
        newState[opindex].g = cost + dist[city][newcity];
        newState[opindex].city = newcity;
        used &= ~(1L << start);
        newState[opindex].h = heuristic(h,g,newcity,depth);
        used |= (1L << start);
        used |= (1L << newcity);
        newState[opindex].used = used;
        used &= ~(1L << newcity);
        opindex++;
      }
      next[opindex-1] = N;
      for(int i=0; i != N; i = next[i]) {
        if (newState[i].g + newState[i].h >= alpha) continue;
        int newtop = stack.top++;
        stack.old[newtop].city = newState[i].city;
        stack.old[newtop].used = newState[i].used;
        stack.old[newtop].g = newState[i].g;
        stack.old[newtop].h = newState[i].h;
        stack.old[newtop].depth = newState[i].depth;
      }
    }
    return alpha;
  }
```

Fazit

Die rekursive Suche löst ein Problem rekursiv bis zur Abbruchbedingung, indem von einem Zustand alle Nachfolgerzustände besucht werden. Dabei wird ein Nachfolger rekursiv verfolgt, bevor ein weiterer besucht wird. Beim Rücksprung müssen vor der Rekursion getroffene Entscheidungen zurückgenommen werden. Dies nennt man Backtracking.

Diskussion (untereinander):

- Welche Probleme sind rücksprungfrei?
- Was können Sie tun, um die Anzahl der rekursiven Aufrufe weiter zu verringern?
- Nennen Sie weitere Einpersonenspiele, die sich durch rekursiver Suche lösen lassen.
- Wie kann ein Programm vom Compiler optimiert werden?