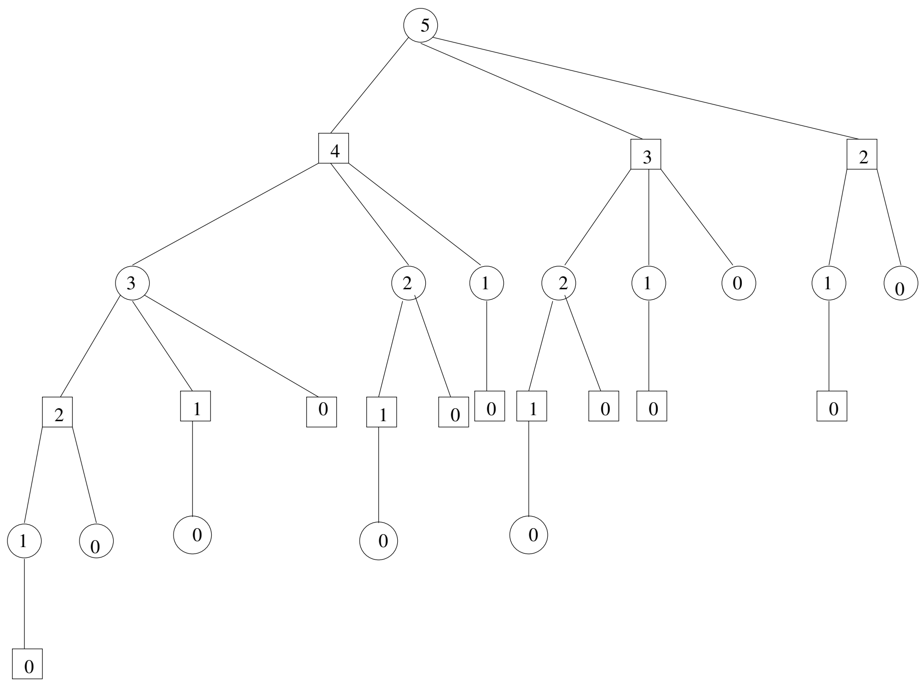
Technische Universität München
Fakultät für Informatik
Dr. M. Schneider, Dr. M. Pizka,
Dr. A. Vilbig, Dr. R. Letz,
G. Bauer, B. Zhang

SS 2003
Lösungsvorschläge zu Blatt 9
20. Juni 2003

Übungen zur Vorlesung Einführung in die Informatik IV

Aufgabe 37 Nimm-Spiele

(a) In dem angegebenen Spielbaum verwenden wir Quadrate und Kreise als Knotensymbole, um kennzuzeichnen, welcher Spieler gerade am Zug ist. Dann ergibt sich folgender Spielbaum:



(b) Offensichtlich ist ein Spieler immer dann in einer Gewinnposition, wenn für die Zahl s der vorhandenen Streichhölzer gilt: $1 \leq s \leq n$. Darauf aufbauend läßt sich rekursiv definieren, ob eine Position eine Gewinnposition ist: Eine Spielsituation ist durch die Zahl s der Hölzer gekennzeichnet. Ein Spieler ist bei s Streichhölzern in einer Gewinnpositon, genau dann, wenn er am Zug ist und wenn

- entweder $1 \leq s \leq n$
- oder wenn $s > n$ und es ein s' gibt mit der Eigenschaft $s - n \leq s' \leq s - 1$, so dass alle s'' mit der Eigenschaft $s' - n \leq s'' \leq s' - 1$ Gewinnstellungen sind.

Als Prädikat ausgedrückt erhalten wir also:

$$s \text{ ist Gewinnstellung } \Leftrightarrow (0 < s \leq n) \vee$$
$$(s > n) \wedge (\exists\, s' : (s - n \leq s' \leq s - 1) \wedge (\forall s'' : (s' - n \leq s'' \leq s' - 1) \Rightarrow \text{ s}'' \text{ ist Gewinnstellung}))$$

Anschaulich ausgedrückt handelt es sich bei s also um eine Gewinnstellung, wenn es einen Zug gibt, so dass der Gegner durch keinen Zug das Spiel zu seinen Gunsten beeinflußen kann. Die Existenz- bzw. Allquantoren in obigem Prädikat spiegeln wieder welcher Spieler gerade am Zug ist: Der Existenzquantor steht dafür, dass der Spieler am Zug ist, für den eine optimale Spielstrategie gesucht wird; der Allquantor dagegen spiegelt die Tatsache wieder, dass der Gegner alle mögliche Spielzüge durchführen kann.

(c) Durch Breitensuche ermitteln wir nun für obige Ausgangssituation (d.h. s = 5 und n = 3) die günstige Spielstrategie: Da s = 5 ist $s' \in \{4,3,2\}$. Nun muß für jedes s' die Menge der möglichen Werte s'' ermittelt werden. Dies sind $S''_4 = \{3,2,1\}$ für $s' = 4$, $S''_3 = \{2,1,0\}$ für $s' = 3$ und $S''_2 = \{1,0\}$ für $s' = 2$; Eine Spielsituation s ist nun Gewinnsituation, wenn es eine Menge S'' gibt, die nur aus Gewinnsituationen besteht. Dies trifft nur für die Menge S''_4 zu. Optimale Strategie ist deshalb die Wegnahme eines einzigen Streichholzes.

(d) Angabe einer rekursiven Prozedur zur Ermittlung, ob Ausgangssituation Gewinnstellung ist:

```
Gewinnstellung(s, n)
if 1 ≤ s ≤ n then
   return w
elsif s ≤ 0 then
   return f
else value := f ;
   forall 1 ≤ i ≤ n
     if Gewinnstellung(s − i, n) = f
       then value := w
     fi;;
     return value
   end
fi
```

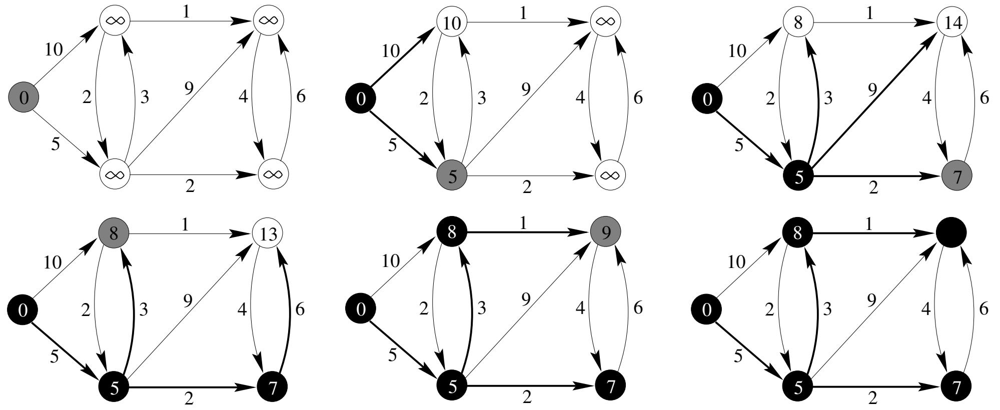
Das Verfahren lässt sich natürlich noch dahingehend optimieren, dass die for-Schleife abgebrochen werden kann, wenn *value* bereits **w** ist.

Aufgabe 38 Algorithmus von Dijkstra

Gegeben sei ein (gerichteter oder ungerichteter) Graph mit Knotenmenge `V` und positiven Kantengewichten `d: V × V → ℕ ∪ {∞}`.

(a) Bestimmen Sie mit dem Algorithmus von Dijkstra für den folgenden Graphen die Länge der kürzesten Verbindung vom Knoten s zu allen anderen Knoten.

Die Schritte des Algorithmus sind nachfolgend dargestellt. In den Knoten sind die kürzesten Entfernungen `l(v)` eingetragen. Dabei sind die Knoten mit geringster Entfernung jeweils grau und bereits abgearbeitete Knoten schwarz hinterlegt, Die jeweils zur Berechung herangezogenen Kanten sind dicker gezeichnet.

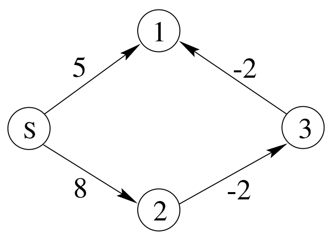


(b) Schätzen Sie die Zeitkomplexität des Algorithmus ab.

Das Durchlaufen der äußeren Schleife liefert den Faktor $O(n)$, wobei n die Anzahl der Knoten im Graph ist. In der Inneren Schleife hat das Suchen nach dem Minimum Komplexität $O(n)$ und das Aufsuchen aller Nachbarn ebenfalls $O(n)$. Daher ist die Komplexität des Dijkstra-Algorithmus $O(n^2)$.

(c) Konstruieren Sie einen Graphen mit ganzzahligen (nicht notwendigerweise positiven) Kantengewichten, für den der Algorithmus von Dijkstra ein falsches Ergebnis liefert.

Es handelt sich bei dem Algorithmus um einen *greedy*-Algorithmus, da immer der Knoten gewählt wird der vom aktuellen Knoten die kürzeste Enfernung hat. Für Graphen mit ganzzahligen Kantengewichten ist die Eigenschaft der optimalen Teillösung des *greedy*-Algorithmus i.A. verletzt. Man kann also einen Fall erzwingen, in dem ein Knoten ausgewählt wird, obwohl er über einen anderen Pfad günstiger zu erreichen ist, wie im folgenden Beispiel:

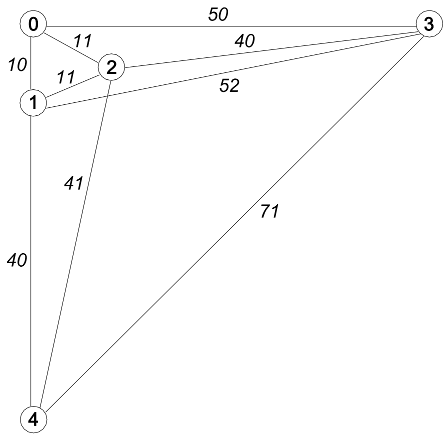


<u>Aufgabe 39</u> (P) Rundreiseproblem, Greedy-Heuristik

Die Greedy-Strategie zur Ermittlung einer möglichst kurzen Rundreise in einem Graphen verspricht folgende Vorteile:

- Gegenüber einer Überprüfung *aller* möglichen Pfade wird der erforderliche Speicher- und Rechenaufwand erheblich reduziert. Während die exakte Lösung eine Komplexität von $O(n!)$ aufweist (es gibt $n!$ verschiedene Permutationen der Knoten-Indizes), benötigt die angegebene Heuristik nur $O(n^2)$ Rechenschritte (jeder der n Knoten wird hinsichtlich seiner Entfernung zu den anderen $n-1$ Knoten des Graphen untersucht).
- Der Algorithmus und seine Implementierung sind besonders einfach (s. Teilaufgabe b).

Allerdings führt die wiederholte Suche nach dem jeweils *lokalen* Optimum, also dem Pfad zum nächstgelegenen Knoten, nicht unbedingt zum *globalen* Optimum, also der insgesamt kürzesten Rundreise durch den Graphen.



[AV: Kante zwischen 0 und 4 wird noch ergänzt]

Beispielsweise führt die Anwendung des Verfahrens im oben gezeigten Beispielgraphen zur Ermittlung einer Rundreise `<0, 1, 2, 3, 4>` mit der Länge $10+11+40+71+50=182$, während die optimale Lösung `<0, 2, 3, 4, 1>` eine Länge von $11+40+71+40+10=172$ aufweist. Intuitiv formuliert, liefert die Heuristik vergleichsweise schlechte Lösungen, falls der Graph sehr unregelmäßig aufgebaut ist, also „Ausreisser“ mit besonders weiter Entfernung zu anderen Knoten besitzt. In diesem Fall wäre ein weiterer „Überblick“ erforderlich, um eine möglichst kurze Rundreise zu bestimmen.

Eine nahe liegende Verbesserung des Greedy-Verfahrens ist daher die geeignete Berücksichtigung von ein oder mehreren Zwischenknoten bei Bestimmung des nächsten kürzesten Pfads. Allerdings steigt mit der Tiefe der so untersuchten Teilgraphen der benötigte Aufwand wieder sehr stark an, bis schließlich kein Vorteil gegenüber dem exakten Verfahren verbleibt. Dementsprechend werden in der Praxis aufwendigere Heuristiken zur Lösung des Rundreiseproblems eingesetzt (s. beispielsweise `http://www.math.princeton.edu/tsp/`).

```
public class TSP {
    ...

    // Part b)

    // Calculate approximation of shortest path by greedy search
    // Returns shortest path as permutation with sequence of node indices
    public Permutation shortestPathGreedy() {
        Permutation spath = new Permutation();
        // Pick random start node
        int start = (int) (Math.random() * numNodes);
        spath.add(start);
        for (int i = 0; i < numNodes - 1; i++) {
            // Select next node with minimal distance
            int from = spath.get(i);
            int minDistance = INF;
            int nearestNeighbor = -1;
            for (int j = 0; j < numNodes; j++) {
                // Only consider nodes not already chosen
                if ((from != j) && !spath.contains(j) &&
                    (distance[from][j] < minDistance)) {
                    minDistance = distance[from][j];
```

```
                nearestNeighbor = j;
            }
        }
        spath.add(nearestNeighbor);
    }
    return spath;
}

// Part c)

// Calculate a random path for arbitrary graphs
// Returns shortest path as permutation with sequence of node indices
public Permutation shortestPathRandom() {
    Permutation spath = new Permutation();
    int start = (int) (Math.random() * numNodes);
    spath.add(start);
    for (int i = 0; i < numNodes - 1; i++) {
        int newNode = start;
        while (spath.contains(newNode))
            newNode = (int) (Math.random() * numNodes);
        spath.add(newNode);
    }
    return spath;
}

// Generate random graph with given number of nodes & max. distance
private void generateRandomGraph(int number, int maxDistance) {
    numNodes = number;
    distance = new int[number][number];
    for (int i = 0; i < number; i++) {
        for (int j = 0; j < number; j++) {
            if (i == j)
                distance[i][j] = 0;
            else
                distance[i][j] = (int) (Math.random() * maxDistance);
        }
    }
}

// Part d)

// Test heuristics with random graphs of growing size
public static void main(String[] args) {
    TSP tsp = new TSP();
    for (int i = 3; i < 10; i++) {
        tsp.generateRandomGraph(i, 10);
        System.out.println("*** Problem size: " + i);
        long t_s = System.currentTimeMillis();
        Permutation spath = tsp.shortestPath();
        long t_f = System.currentTimeMillis();
        long t = t_f - t_s;
```

```
        int length = tsp.pathLength(spath);
        System.out.println("Complete search: " + spath + " with length " +
                           length + " found in " + t + " ms");
        t_s = System.currentTimeMillis();
        spath = tsp.shortestPathGreedy();
        t_f = System.currentTimeMillis();
        t = t_f - t_s;
        length = tsp.pathLength(spath);
        System.out.println("Greedy search:   " + spath + " with length " +
                           length+ " found in " + t + " ms");
        t_s = System.currentTimeMillis();
        spath = tsp.shortestPathRandom();
        t_f = System.currentTimeMillis();
        t = t_f - t_s;
        length = tsp.pathLength(spath);
        System.out.println("Random search:   " + spath + " with length " +
                           length+ " found in " + t + " ms");
    }
}
```