Dr. Michael J. Winckler
IWR, Raum 302
INF 368
69120 Heidelberg
Michael.Winckler@iwr.uni-heidelberg.de
http://www.iwr.uni-heidelberg.de/teaching/Mathe-Star/

Arbeitsblatt 11 *Grafik für Traveling Salesman* (Version 1.01)

Theorie

Übungsziele: Nachdem wir uns im letzten Übungsblatt mit dem TSP beschäftigt haben, stehen nun eine grafische Aufbereitung dieses Modells und einige Suchheuristiken auf dem Plan. Dieses Übungsblatt soll dazu die nötigen Grundlagen vermitteln, aus denen wir dann Klassen und Algorithmen entwickeln können.

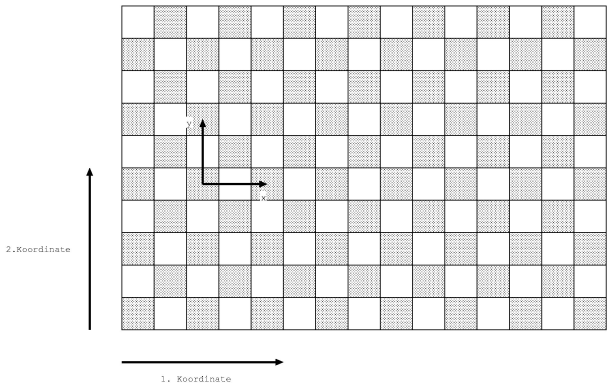
Übungen

Info 11.1

Die bisher von uns verfolgten Ansätze für eine Struktur zur bearbeitung von Bildern hat ausschliesslich auf der Ebene von Pixeln gearbeitet. Dies reicht zwar aus, wenn die Skalierungen der Anwendung sich leicht in Pixel-Koordinaten umrechnen lassen, ist aber auf die Dauer wenig produktiv.

Eine sinnvolle Neuorientierung auf Basis des C++-Klassenkonzeptes sollte daher berücksichtigen, dass eine Grafik stets zwei parallele Koordinatensysteme zur Grundlage hat: zum einen gibt es ein Pixel-Koordinatensystem, zum anderen ein 2D-Koordinatensystem zur Beschreibung des Grafikinhaltes. Beispielsweise ist es für die TSP-Implementierungen zu Problemstellungen im Einheitsquadrat $[0;1] \times [0;1]$ sinnvoll, ein Bild von 800×800 Pixeln zur Darstellung zu verwenden und dabei nur in den inneren 700×700 Pixel die eigentliche Grafik zu erzeugen.

Aus dem „I am alive"-Prinzip der Objektorientierung ergibt sich, dass die Grafikklasse slbst die Umrechnung zwischen den beiden Koordinatensysteme vornehmen kann. Dazu benötigt die Klasse vier Informationen (Daten): Die Koordinaten des 2D-Ursprungs, gemessen in Pixeln und die Laufweite einer x– bzw. y–Einheit, ebenfalls in Pixeln. Während die erste Angabe ein Integerwert sein sollte (auch wenn man sich prinzipiell hier auch `float`–Werte vorstellen kann), ist das zweite Wertepaar ein Float-Wert, um z.B. auch Verkleinerungen möglich zu machen.



Eine weitere Schwierigkeit ist die Tatsache, dass Bildverarbeitungsprogramme meist die Bildkooridnaten

in der linken oberen Ecke beginnen, während unser intuitiver Zugang über Graphen von Funktionen die linke untere Bildkoordinate als Ursprung suggeriert. Dies Diskrepanz müssen wir ein für allemal feststellen und mögliche Unterschiede in die Lade- und Speicherroutinen verlagern: diese sind nämlich unsere Schnittstelle zwischen der von uns implementierten internen Darstellung und der von anderen Grafikformaten verwendeten externen Darstellung. Ein Lösungsvorschlag ist in der Skizze dargestellt.

Wir sehen uns wegen des bisher gesagten genötigt, alle Routinen zur Bearbeitung des Bildes in zwei Varianten zu schreiben: Zum einen in Weltkoordinaten, zum anderen in Pixelkoordinaten. Die Umrechnung zwischen den beiden Koordinatensystemen ist allerdings einfach, da linear:

$$\begin{aligned} pix_x &= pix_{x_0} + welt_x * d_x \\ pix_y &= pix_{y_0} + welt_y * d_y \end{aligned}$$

sowie

$$\begin{aligned} welt_x &= (pix_x - pix_{x_0})/d_x \\ welt_y &= (pix_y - pix_{y_0})/d_y \end{aligned}$$

sind die beiden Umrechenformeln, wobei pix_{x_0}, pix_{y_0} die Pixelkoordinaten des Weltkoordinatenursprungs sind und d_x, d_y die Anzahl der Pixel, der eine Weltkoorinateneinheit entspricht. In dem oben erwähnten Beispiel für TSP im Einheitsquadrat ergeben sich:

$$\begin{aligned} pix_{x_0} &= 50 \\ pix_{y_0} &= 50 \\ d_x &= 700 \\ d_y &= 700. \end{aligned}$$

Info 11.2

Zu den bisher besprochenen Suchheuristiken fehlt uns noch eine Systematisierung. Hier folgen nun vier einfache Suchalgorithmen, von denen wir einen schon implementiert haben:

Nearest Next Element Algorithm

Beginnend mit einem einzelnen Vertex („Element") wird unter allen noch nicht besuchten Vertices derjenige ausgesucht, der dem aktuell letzten besuchten Knoten am nächsten liegt. Dieser Knoten wird als nächster in die Liste aufgenommen.

1. Starte mit einer Tour, die aus einem beliebigen Vertex v_i besteht. Dies ist das erste Endvertex $\bar{v}$.
2. Wenn es noch freie Vertices gibt, suche den freien Vertex v_k mit minimalem Abstand zwischen v_k und $\bar{v}$.
3. Füge v_k nach $\bar{v}$ ein und mache es zum neuen Endvertex $\bar{v}$.

Nearest Addition Algorithm

Während *Nearest next element* nicht berücksichtigt, dass man beim hinzufügen des k-ten Elementes schon einen Pfad zwischen $k-1$ Elementen hat, wird diese hier nun ausgenutzt:

1. Starte mit einer Tour, die aus einem Vertex besteht.
2. Wenn es noch freie Vertices gibt, suche den freien Vertex v_k und den Tour-Vertex v_j mit minimalem Abstand $c_{j,k}$.
3. Wähle v_i, einen der beiden Nachbarknoten von v_j und füge v_k zwischen v_i und v_j ein.

Nearest Insertion Algorithm

Eine kleine Verbesserung erhält man, indem man den Einfügeprozess verfeinert:

1. Starte mit einer Tour, die aus einem Vertex besteht.
2. Wenn es noch freie Vertices gibt, suche den freien Vertex v_k und den Tour-Vertex v_j mit minimalem Abstand $c_{j,k}$.
3. Suche nun innerhalb der gesamten Tour den besten Einfügeplatz v_m, v_n von aufeinanderfolgenden Staedten, sodass $c_{m,k} + c_{k,n} - c_{m,n}$ am kleinsten wird.

Cheapest Insertion Algorithm

Eine weitere Verbesserung erhält man, indem man gleich diejenige Tour einfügt, deren Einfügekosten am geringsten sind. Dazu muss Schritt 3 schon zusammen mit Schritt 2 bearbeitet werden:

1. Starte mit einer Tour, die aus einem Vertex besteht.
2. Wenn es noch freie Vertices gibt, suche den freien Vertex v_k und den Einfügepunkt v_m, v_n, sodass $c_{m,k} + c_{k,n} - c_{m,n}$ am kleinsten wird.
3. Füge v_k zwischen v_m und v_n ein.