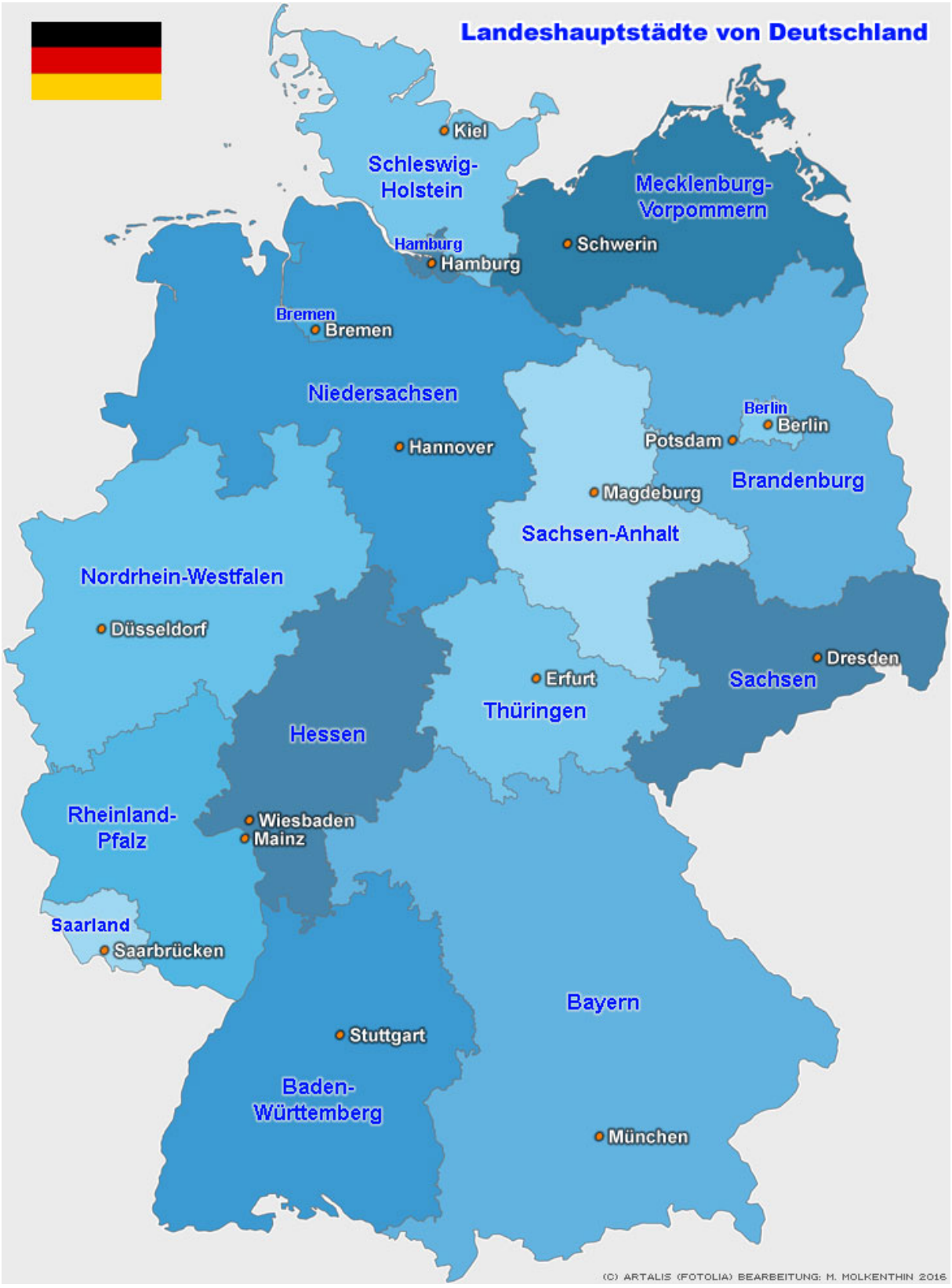
Miniprojekt 1

Problembeschreibung

Diese Aufgabe beschäftigt sich mit dem Problem des Handlungsreisendens. Das Ziel der Aufgabe ist Dich mit Optimierungsalgorithmen vertraut zu machen und Dich beobachten zu lassen, wie effektiv Optimierungsalgorithmen für aufwändige Aufgaben sein können.

Ein Handlungsreisender möchte seine Kunden in verschiedenen Städten besuchen. Um die Kosten der Reise gering zu halten, möchte er die Reisestrecke möglichst kurz halten. Er plant eine Rundreise. Bei dieser startet er von seiner Heimatstadt und steuert dann die Städte **aller** Kunden genau **einmal** an. Anschließend kehrt er zum Startpunkt zurückkehrt. Für die Planung der Rundreise benötigt der Handlungsreisende damit einer Liste der Städte und eine Tabelle der Reiseentfernungen zwischen den Städten. Da die Reiseentfernungen von A nach B und von B nach A nicht unbedingt gleich sein müssen, handelt es sich hier um das Asymmetrische TSP (ATSP). In vorliegendem Fall gehen wir davon aus, dass der Handlungsreisende seine Kunden in allen Landeshauptstädten der Bundesrepublik besuchen möchte.

	Kiel	Hamburg	Schwerin	Potsdam	Berlin	Dresden	Muenchen	Stuttgart	Mainz	Saarbruecken	Duesseldorf	Hannover	Bremen	Magdeburg	Erfurt	Wiesbaden
Kiel	0	97	151	349	354	542	886	750	617	766	491	254	210	376	486	612
Hamburg	97	0	111	283	288	476	790	654	521	670	406	158	125	280	391	516
Schwerin	150	111	0	208	213	401	758	757	624	773	510	262	229	212	474	620
Potsdam	350	283	208	0	35	202	555	602	545	694	532	257	365	127	271	541
Berlin	355	288	213	35	0	193	584	631	574	723	561	286	406	156	300	570
Dresden	543	477	401	203	192	0	459	506	488	637	583	365	472	227	214	484
Muenchen	886	791	757	554	584	461	0	232	426	440	615	659	767	521	413	423
Stuttgart	751	655	805	602	632	508	233	0	213	222	414	512	631	509	339	219
Mainz	617	522	624	545	575	491	431	213	0	147	221	379	470	453	286	16
Saarbruecken	766	671	773	694	724	639	440	217	150	0	278	527	551	602	435	159
Duesseldorf	487	402	505	529	559	582	613	406	217	298	0	278	292	421	404	203
Hannover	246	151	253	254	284	365	657	512	379	528	281	0	127	147	248	374
Bremen	211	127	229	364	406	475	767	630	472	576	298	135	0	257	367	467
Magdeburg	376	281	212	125	155	232	523	511	454	603	424	149	257	0	171	450
Erfurt	488	392	475	271	301	217	413	340	286	435	415	249	369	169	0	281
Wiesbaden	613	518	620	541	571	487	423	214	16	158	206	375	467	449	282	0


Landeshauptstädte von Deutschland
Kiel
Schleswig-Holstein
Mecklenburg-Vorpommern
Schwerin
Hamburg
Hamburg
Bremen
Bremen
Niedersachsen
Hannover
Berlin
Berlin
Potsdam
Brandenburg
Magdeburg
Sachsen-Anhalt
Nordrhein-Westfalen
Düsseldorf
Dresden
Sachsen
Erfurt
Thüringen
Hessen
Rheinland-Pfalz
Wiesbaden
Mainz
Saarland
Saarbrücken
Bayern
Stuttgart
Baden-Württemberg
München
(C) ARTALIS (FOTOLIA) BEARBEITUNG: M. MOLKENTHIN 2016

Miniprojekt 1

Aufgabe 1 (Brute Force)

Helfe dem Handlungsreisenden, indem Du ein Programm schreibst, das alle möglichen Rundreisen berechnet und die kürzeste Rundreise sowie die Rechenzeit zum Schluss ausgibt.

1. Dein Programm sollte den Parameter n haben, mit dem man angeben kann, wie viele Städte der Handlungsreisende besuchen möchte. Bei $n = 3$ würde der Handlungsreisende nach der kürzesten Rundreise suchen, die die Städte Kiel (1), Hamburg (2) und Schwerin (3) enthält.

2. Messe die Längen der kürzesten Reisen und die Rechenzeiten für $n = 5, 6, 7, 8, 9, 10, ...$ Du kannst das n solange erhöhen, bis die Rechenzeit für deinen Computer zu groß wird. Trage die Werte in die unten stehende Tabelle ein (Spalten: Brute Force).

3. Extrapoliere die Rechenzeit für $n = 16, 15, ...$ und trage die extrapolierte Werte auch in die Tabelle ein. Könnte Dein "Brute Force" Programm in diesem Jahr die kürzeste Rundreise durch alle Landeshauptstädte berechnen?

Hinweis: Du kannst für die Erzeugung aller Permutation der Elemente einer Liste die Python-Bibliothek `itertools` verwenden. Beachte, dass `itertools` auch redundante Permutationen erzeugt, wie beispielsweise [1, 2, 3], [2, 3, 1], [3, 1, 2]. Überlege Dir aber, ob die Vermeidung der Längenberechnungen für redundante Rundreisen zeitlich nicht Aufwändiger ist, als der zeitliche Mehraufwand für redundante Längenberechnungen. Eine Anleitung für die Generierung von Permutationen mit itertools findest Du unter: `http://docs.python.org/library/itertools.html#recipes`.)

■

Miniprojekt 1

	Brute Force		2-opt								SA									
n	Len	Rechenzeit	Len	Len	Len	Len	Len	$\overline{Len}$	$\overline{\frac{Berechnete}{Rundreisen}}$	$\overline{\frac{Rechen-}{zeit}}$	Len	Len	Len	Len	Len	$\overline{Len}$	$\overline{\frac{Berechnete}{Rundreisen}}$	$\overline{\frac{Rechen-}{zeit}}$
5																		
6																		
7																		
8																		
9																		
10																		
11																		
12																		
13																		
14																		
15																		
16																		

Aufgabe 2 (2-opt Improvement Heuristic)

Da der Handlungsreisende alle Landeshauptstädte besuchen möchte, fragt er Dich, ob Du für $n = 16$ eine Rundreise berechnen könntest. Die Rundreise muss nicht optimal sein, aber doch so kurz wie möglich. Der Handlungsreisende wäre mit einer Antwort, die in einigen Minuten berechenbar ist, zufrieden.

Schreibe eine 2-opt Heuristik. Teile Deine Implementierung in drei Funktionen auf.

1. Implementiere die Funktion `two_opt_xchg(tour, I, k)`, die die Reihenfolge der Städte $i, ..., k$ in `tour` umdreht und die neue Rundreise zurückgibt.

2. Implementiere danach die Funktion `two_opt_step(tour)`, die die `best-improvement` Strategie realisiert. Dafür muss `two_opt_step` alle sinnvollen 2-opt Vertauschungen von `tour` durchführen und die kürzeste neue Rundreise sich merken sowie zurückgeben. Es sollte beachtet werden, dass ein 2-opt Tausch (`two_opt_xchg`) bei der `best-improvement` Strategie immer auf die original-`tour` angewendet wird. Zähle auch die Anzahl der ausgewerteten Rundreiselängen und lasse auch diese Größe zurückgeben.

3. Implementiere einen Hill Climber, der, ausgehend von einer zufälligen Startlösung, `two_opt_step(tour)` solange aufruft, solange eine Verbesserung der Rundreise möglich ist. Kann die Rundreise nicht mehr verbessert werden, lasse die Länge der Rundreise, die Anzahl berechneter Rundreiselängen und die Rechenzeit ausgeben.

4. Berechne für $n = 5, ..., 16$ die Rundreisen mit 2-opt.

5. Da 2-opt ein nichtdeterministischer (randomisierter) Algorithmus ist, lasse ihn fünf Mal laufen und berichte die Rundreiselängen, die durchschnittliche Anzahl der Berechnungen einer Rundreiselänge und die durchschnittliche Rechenzeit. Trage die Werte in die Tabelle ein.

6. Für bekannte kürzeste Rundreisen: Findet 2-opt auch immer die kürzeste Rundreise? Wenn nicht, wie nahe kommt 2-opt an eine beste Lösung?

7. Nimmt die 2-opt Rechenzeit mit größer werdenden n zu?

■

Aufgabe 3 (Simulated Annealing)

Der Handlungsreisende fragt Dich, ob die die Rundreiselängen und Rechenzeiten von 2-opt verbessern kannst. Du entscheidest Dich Simulated Annealing (SA) eine Chance zu geben.

1. Implementiere den SA Algorithmus.
2. Der Perturbationsoperator (OP) soll als ein Kombinationsoperator implementiert werden, der bei jedem Aufruf zufällig eine der folgenden Operationen durchführt
   - ein Schritt des Algorithmus `node exchange`,
   - ein Schritt des Algoirthmus `node insertion` oder
   - ein Schritt des Algorithmus `two_opt_xchg(tour, i, k)`.
3. Stelle einige Experimente mit verschiedenen Abkühlschemas und deren Parametern an und wähle die beste Kombination.
4. Lasse SA fünf Mal laufen und berichte die Rundreiselängen, die durchschnittliche Anzahl der Berechnungen einer Rundreiselänge und die durchschnittliche Rechenzeit. Trage die Werte in die Tabelle ein.
5. Für bekannte kürzeste Rundreisen: Findet SA auch immer die kürzeste Rundreise? Wenn nicht, wie nahe kommt SA an eine beste Lösung?
6. Nimmt die Rechenzeit von SA mit größer werdenden n zu?
7. Ist Deine SA Implementierung schneller als 2-opt?
8. Erzeugt Dein SA bessere Lösungen als 2-opt?

■

General Remarks

Wrong way to copy a list in Python:

```
>>> a=[1,2,3]
>>> b=a       % this is just defining a second name for the same list
>>> b[1]="Hallo Welt"
>>> print(a)
[1, 'Hallo Welt', 3]
```

Correct way to copy a list in Python:

```
>>> a=[1,2,3]
>>> b=list(a) % this is a real copy/clone of the list a
>>> b[1]=99
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 99, 3]
```

If `two_opt_xchg(newTour, distances)` doesn't make a real copy of tour, it would modify the variable `tour`, i.e. `tour` and `newTour` are two different names for the same list:

```
newTour = two_opt_xchg(tour,i,k)
newDistance = measurePath(newTour,distances)
if newDistance < shortestLength:
    shortestTour = newTour
    shortestLength = newDistance
```

This is correct:

```
newTour = two_opt_xchg(list(tour),i,k) % make a real copy of tour
                                       % before modifying the copy
newDistance = measurePath(newTour,distances)
if newDistance < shortestLength:
    shortestTour = list(newTour) % <-- This is also important
    shortestLength = newDistance
```