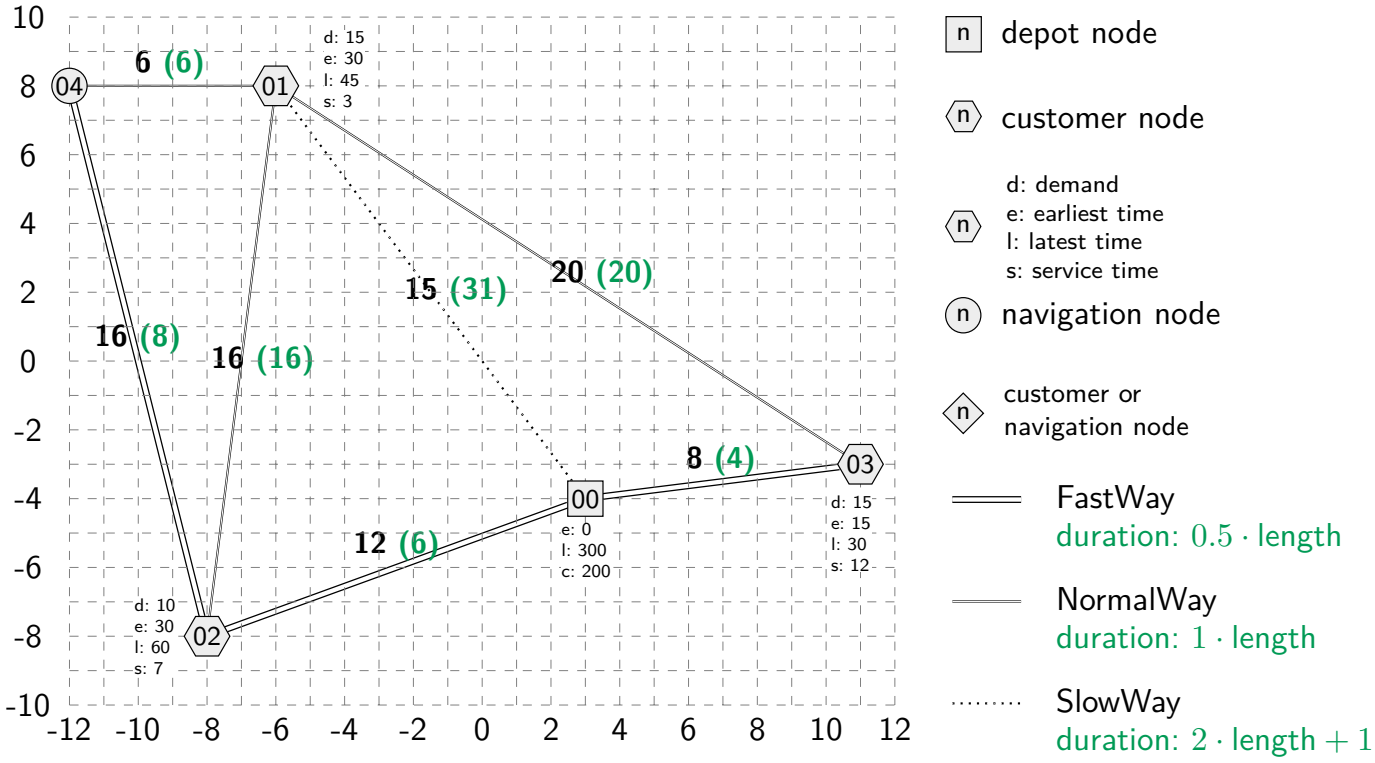
Das zu modellierende Problem



Die Bedarfe, Zeitfenster und Servicezeiten können in der Karte ebenfalls vermerkt werden.

Das Grundgerüst / die Präambel

```
1  public class WetterauOrders {    ❶
2
3    public static final int UNITS = 0;    ❷
4
5    public static void main(String[] args){    ❸
6      VehicleRoutingProblem.Builder vrpBuilder =
7        VehicleRoutingProblem.Builder.newInstance();    ❹
8    }
9
10 }
```

❶ Das Gerüst für ein JSprit-Programm bildet eine gewöhnliche Java-Klasse.

❷ Da in diesem Beispiel alle Kundenbedarfe in Kapazitätseinheiten umgerechnet wurden, werden in dem Programm die Bedarfe auch in Einheiten angegeben.

❸ Die Probleme werden innerhalb der `main`-Methode programmiert.

❹ Ausgangspunkt für die Programmierung eines Problems ist eine `VehicleRoutingProblem`-Instanz. JSprit verwendet hier das sog. Builder-Pattern (cf. [Blo17]), das u.a. dafür sorgt, dass ein Objekt der entsprechenden Klasse in einem konsistenten Zustand ist.

Programmierung der Knoten und Kunden

```
1  public static void main(String[] args){
     :
     :
10   Location n0 = Location.newInstance(2, 4);  (1)
11
12   Service n1 = Service.Builder.newInstance("n1")  (2)
13     .setLocation(Location.newInstance(-6, 8))  (3)
14     .addSizeDimension(UNITS, 15)  (4)
15     .setTimeWindow(TimeWindow.newInstance(30, 45))
16     .setServiceTime(3)
17     .build();
18
19   // Service n2, n3 genauso
20   // Location n4 (Navigationsknoten) wie n0
21 }
```

1. Knoten werden generell als `Location`-Instanzen programmiert.
2. Kunden werden als `Service`-Instanzen programmiert.
3. Auch hier wird eine `Location`-Instanz benötigt, womit sich ein Kunden-Knoten ergibt.
4. Der erste Kunde unseres Beispiels hat einen Bedarf von 15 Einheiten.

Programmierung der Knoten und Kunden

```
1  public static void main(String[] args){
     :
     :
10   Location l0 = Location.newInstance(2, 4);  ❶
11
12   Service n1 = Service.Builder.newInstance("n1")  ❷
13     .setLocation(Location.newInstance(-6, 8))  ❸
14     .addSizeDimension(UNITS, 15)  ❹
15     .setTimeWindow(TimeWindow.newInstance(30, 45)) ❺
16     .setServiceTime(3) ❻
17     .build(); ❼
18
19   // Service n2, n3 genauso
20   // Location n4 (Navigationsknoten) wie n0
21 }
```

❺ Das Zeitfenster öffnet zum Zeitpunkt 30 und schließt zum Zeitpunkt 45.

❻ Für den Kunden wurde eine Service-Dauer von 3 Zeiteinheiten berechnet.

❼ Dies ist Teil des Builder-Patterns: Durch den Aufruf von `build()` wird die eigentliche Service-Instanz erzeugt.

Programmierung der Knoten und Kunden

```
5  public static void main(String[] args){
     VehicleRoutingProblem.Builder vrpBuilder = ...

     Location n0 = ...

     Service n1 = ...

     Service n2 = ...

     Service n3 = ...

     Location n4 = ...

     .
     .
     .
34
35   vrpBuilder.addAllJobs(Arrays.asList(n1, n2, n3));  ❶
36 }
```

❶ Die Kunden müssen als solche (genauer: als `Jobs`) dem Builder für das `VehicleRoutingProblem` hinzugefügt werden.

Programmierung der Zeitfunktionen 1

```
5   public static void main(String[] args){
    :
    :
122 }
123
124 public static double fastWayFunction(Location end1, Location end2) { (1)
125   return 0.5 * EuclideanDistanceCalculator
126     .calculateDistance(end1.getCoordinate(), end2.getCoordinate());
127 }
128
129 public static double normalWayFunction(Location end1, Location end2) { (1)
130   return EuclideanDistanceCalculator
131     .calculateDistance(end1.getCoordinate(), end2.getCoordinate());
132 }
133
134 public static double slowWayFunction(Location end1, Location end2) { (1)
135   return 2 * EuclideanDistanceCalculator
136     .calculateDistance(end1.getCoordinate(), end2.getCoordinate()) + 1;
137 }
```

(1) Für jeden Straßentyp, für den eine eigene Berechnung der Fahrtzeit erforderlich ist, wird eine statische Methode definiert. Diese Methode erwartet als Parameter die jeweiligen `Locations` der durch die Kante verbundenen Knoten.

Programmierung der Zeitfunktionen 2

- Für das Hinzufügen der eigentlichen Kanten stehen zwei unterschiedliche Herangehensweisen zur Verfügung.
- Die erste Variante eignet sich für Straßen-Typen (Kanten-Typen), von denen es nur sehr wenige Exemplare gibt.
- Die zweite Variante hat zunächst einen gewissen Overhead an Code, spart aber ab einer gewissen Anzahl an Kanten Code-Zeilen ein.

Programmierung der Zeitfunktionen 3

```
5  public static void main(String[] args){
   :
   :
37   IncompleteCostMatrix.Builder costMatrixBuilder =  (1)
38             IncompleteCostMatrix.Builder.newInstance();
39
40   // First variant used for SlowWays
41   costMatrixBuilder.addTransportTime(n0,n1.getLocation(),
42     slowWayFunction(n0,n1.getLocation()));
43
44   // Second Variant used for NormalWays
45   Set<RelationKey> normalWays = new HashSet<>();
46   normalWays.add(RelationKey.newKey(n1, n4));
47   normalWays.add(RelationKey.newKey(n1, n2));
48   normalWays.add(RelationKey.newKey(n1, n3));
49
50   for(RelationKey key : normalWays)
51     costMatrixBuilder.addTransportTime(key.from, key.to,
52                                normalWayFunction(key.from, key.to));
53 }
```

(1) Eine Instanz der Klasse `IncompleteCostMatrix` speichert sowohl die Fahrtzeiten als auch die Distanzen. Auch hier wird wieder das `Builder`-Pattern verwendet.[1]

[1] Die Klasse ist kein Original-Bestandteil der JSprit-Bibliothek, sondern wurde von den Athos-Entwicklern ergänzt.

Programmierung der Zeitfunktionen 4

```
5  public static void main(String[] args){
   :
   :
37   IncompleteCostMatrix.Builder costMatrixBuilder =  (1)
38             IncompleteCostMatrix.Builder.newInstance();
39
40   // First variant used for SlowWays
41   costMatrixBuilder.addTransportTime(n0,n1.getLocation(),  (2)
42     slowWayFunction(n0,n1.getLocation()));  (3)
43
44   // Second Variant used for NormalWays
45   Set<RelationKey> normalWays = new HashSet<>();
46   normalWays.add(RelationKey.newKey(n1, n4));
47   normalWays.add(RelationKey.newKey(n1, n2));
48   normalWays.add(RelationKey.newKey(n1, n3));
49
50   for(RelationKey key : normalWays)
51     costMatrixBuilder.addTransportTime(key.from, key.to,
52                               normalWayFunction(key.from, key.to));
53 }
```

(2) Durch den Eintrag der beiden Knoten, entsteht eine Kante zwischen eben diesen Knoten.

(3) Als Fahrtzeit wird der Rückgabewert der `slowWayFunction()` eingetragen. Diese benötigt allerdings bei ihrem Aufruf die beiden Knoten nocheinmal als Parameter.

Programmierung der Zeitfunktionen 5

```
5  public static void main(String[] args){
   :
   :
37   IncompleteCostMatrix.Builder costMatrixBuilder =  (1)
38             IncompleteCostMatrix.Builder.newInstance();
39
40   // First variant used for SlowWays
41   costMatrixBuilder.addTransportTime(n0,n1.getLocation(),  (2)
42     slowWayFunction(n0,n1.getLocation()));  (3)
43
44   // Second Variant used for NormalWays (4)
45   Set<RelationKey> normalWays = new HashSet<>();
46   normalWays.add(RelationKey.newKey(n1, n4));
47   normalWays.add(RelationKey.newKey(n1, n2));
48   normalWays.add(RelationKey.newKey(n1, n3));
49
50   for(RelationKey key : normalWays)  (5)
51     costMatrixBuilder.addTransportTime(key.from, key.to,
52                                normalWayFunction(key.from, key.to));
53 }
```

(4) In der zweiten Variante werden zunächst die Knoten, für die eine Kante erzeugt werden soll, als Paar einer Relation in einer Set-Instanz eingefügt.

(5) Der Eintrag in die Kostenmatrix (bzw. deren Builder) erfolgt dann in einer for-Schleife, die alle Paare der Menge einträgt und die passende Funktion aufruft.

Programmierung der Zeitfunktionen 6

```
5  public static void main(String[] args){
   :
   :
37   IncompleteCostMatrix.Builder costMatrixBuilder =  (1)
38             IncompleteCostMatrix.Builder.newInstance();
39
40   // First variant used for SlowWays
41   costMatrixBuilder.addTransportTime(n0,n1.getLocation(),  (2)
42     slowWayFunction(n0,n1.getLocation()));  (3)
43
44   // Second Variant used for NormalWays (4)
45   Set<RelationKey> normalWays = new HashSet<>();
46   normalWays.add(RelationKey.newKey(n1, n4));
47   normalWays.add(RelationKey.newKey(n1, n2));
48   normalWays.add(RelationKey.newKey(n1, n3));
49
50   for(RelationKey key : normalWays)  (5)
51     costMatrixBuilder.addTransportTime(key.from, key.to,
52                                normalWayFunction(key.from, key.to));
53 }
```

- Als Distanz wird übrigens automatisch der Euklidische Abstand zwischen den beiden Knoten verwendet, weshalb lediglich die Fahrtdauer eingetragen werden muss.

Programmierung der Zeitfunktionen 6

```
5  public static void main(String[] args){
   :
   :
62   costMatrixBuilder.completeTransortTimeMatrix(); (1)
63   IncompleteCostMatrix cm = costMatrixBuilder.build(); (2)
64   vrpBuilder.setRoutingCost(cm); (3)
65 }
```

1. Der Aufruf dieser Methode erzeugt aus einem unvollständigen Fahrtzeit-Graphen einen vollständigen Fahrtzeit-Graphen. Gleichzeitig löst dieser Aufruf die Erzeugung eines vollständigen Distanz-Graphen aus.
2. Auch hier kommt wieder das Builder-Pattern zum Einsatz: Erst durch den Aufruf der `build()`-Methode entsteht die eigentliche `IncompleteCostMatrix`-Instanz.
3. Die `IncompleteCostMatrix`-Instanz muss dem Builder des `VehicleRoutingProblems` hinzugefügt werden.

Programmierung der Vehikel

```
5  public static void main(String[] args){
   :
   :
62   // Vehicle type definition
63   VehicleType vehicleType = VehicleTypeImpl.Builder.newInstance("unitVehicleType") ❶
64     .addCapacityDimension(UNITS, 200).build(); ❷
65
66   // Vehicle instance defintion
67   VehicleImpl vehicleInstance = VehicleImpl.Builder.newInstance("unitVehicleInstance")
68     .setType(vehicleType)
69     .setStartLocation(n0)
70     .build();
71
72     // Adding vehicle instance to the problem
73     vrpBuilder.addVehicle(vehicleInstance);
74 }
```

❶ Als erstes definieren wir den **Typ** des Vehikels. Genutzt wird wieder das Builder-Pattern.

❷ In unserem Beispiel wird über den Typ des Vehikels lediglich die Kapazität der Fahrzeuge festgelegt und schließlich mit dem `build()`-Aufruf der eigentliche Fahrzeugtyp erzeugt.

Programmierung der Vehikel

```
5  public static void main(String[] args){
   :
   :
62   // Vehicle type definition
63   VehicleType vehicleType = VehicleTypeImpl.Builder.newInstance("unitVehicleType") (1)
64     .addCapacityDimension(UNITS, 200).build(); (2)
65
66   // Vehicle instance defintion
67   VehicleImpl vehicleInstance = VehicleImpl.Builder.newInstance("unitVehicleInstance") (3)
68     .setType(vehicleType) (4)
69     .setStartLocation(n0) (5)
70     .build();
71
72     // Adding vehicle instance to the problem
73     vrpBuilder.addVehicle(vehicleInstance);
74 }
```

(3) Nach dem Typ definieren wir nun eine **Instanz** dieses Typs.

(4) Durch Setzen des Typs wird dieser der Instanz zugeordnet.

(5) Der Startpunkt der Ausprägung. Erst hierdurch wird die `Location` n_0 tatsächlich zu einem Depot! Ebenfalls zu beachten: Die Instanz repräsentiert eine unendliche Flotte von Vehikeln.

Programmierung der Vehikel

```
5  public static void main(String[] args){
   :
   :
62   // Vehicle type definition
63   VehicleType vehicleType = VehicleTypeImpl.Builder.newInstance("unitVehicleType") (1)
64     .addCapacityDimension(UNITS, 200).build(); (2)
65
66   // Vehicle instance defintion
67   VehicleImpl vehicleInstance = VehicleImpl.Builder.newInstance("unitVehicleInstance") (3)
68     .setType(vehicleType) (4)
69     .setStartLocation(n0) (5)
70     .build(); (6)
71
72     // Adding vehicle instance to the problem
73     vrpBuilder.addVehicle(vehicleInstance); (7)
74 }
```

(6) Durch den `build()`-Aufruf entsteht das `VehicleImpl`-Objekt

(7) Das `VehicleImpl`-Object wird dem `VehicleRoutingProblem` hinzugefügt.

Erstellen des Problems

```
5  public static void main(String[] args){
   :
   :
62   VehicleRoutingProblem vrp = vrpBuilder.build(); ❶
63
64 }
```

❻ Abschließend wird die `build()`-Methode des `VehicleRoutingProblem`-Builders aufgerufen. Hiermit ist das Problem komplett programmiert/modelliert. Die Aufgaben enden mit dem Modellieren des Problems.

Lösen des Problems[2]

```
5  public static void main(String[] args){
   :
   :
62   VehicleRoutingProblem vrp = vrpBuilder.build();
63
64   VehicleRoutingAlgorithm vra =  ❶
65     Jsprit.Builder.newInstance(vrp)
66       .setProperty(Jsprit.Parameter.FAST_REGRET, "true")
67       .buildAlgorithm();
68
69     vra.setMaxIterations(6000);
70
71     Collection<VehicleRoutingProblemSolution> solutions = vra.searchSolutions(); ❷
72     VehicleRoutingProblemSolution best = Solutions.bestOf(solutions); ❸
73     SolutionPrinter.print(vrp,best,Print.VERBOSE); ❹
74 }
```

❻ Das modellierte Problem wird dem Builder für eine `VehicleRoutingAlgorithm`-Instanz übergeben. Es folgt das Setzen einiger algorithmusspezifischer Eigenschaften.

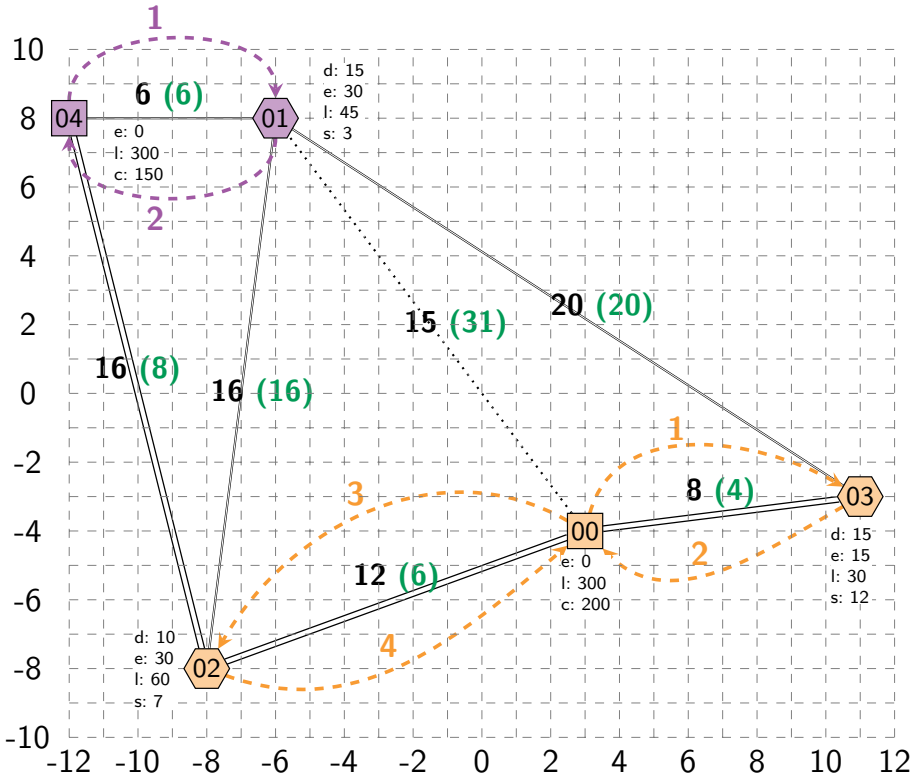
❼ Der Algorithmus erstellt den Lösungsraum.

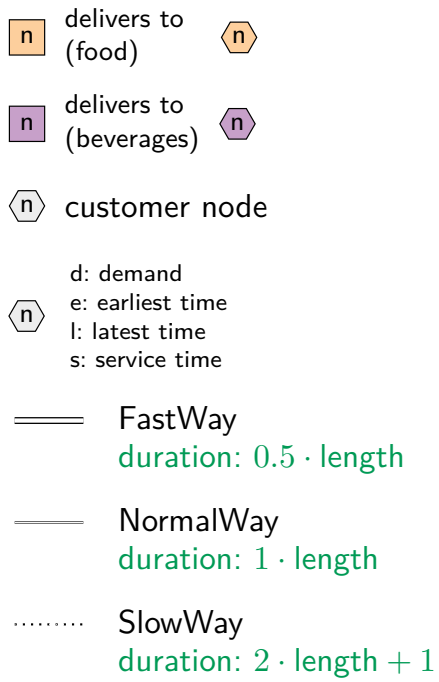
❽ Die beste Lösung im Hinblick auf die Zielfunktion wird ausgewählt.

❾ Diese Lösung wird auf der Konsole ausgegeben.

[2] Wird in den Aufgaben nicht geprüft!

Änderung der Geschäftsstrukturen





Neues Szenario: Getränkemarkt bei Knoten 04, Kunde 01 braucht Getränke. Knoten 00 liefert weiter Lebensmittel, die von den Kunden 02 und 03 bestellt werden.

Änderung der Geschäftsstrukturen

- Viele Elemente des bisherigen Programms können einfach übernommen werden.
  - `Locations` bleiben unverändert.
  - Definition der Kanten und der Fahrtzeiten bleiben identisch.
- Die folgenden Änderungen müssen vorgenommen werden:
  - Die `UNITS = 0` werden nun durch `UNITS_FOOD = 0` und `UNITS_BEVERAGE = 1` ersetzt.
  - `Service.Builder` wird zu `Delivery.Builder` und `Service` zu `Delivery`
  - Der generelle `VehicleType` wird durch zwei spezielle `VehicleTypes` ersetzt, die Kapazitäten für das entsprechende Produkt haben und in der jeweiligen `Location` starten.

Lösen des Problems

```
1  public class SmallWetterau2 {
2    public static final int UNITS_FOOD = 0;
3    public static final int UNITS_BEVERAGE = 1;
4
5    public static void main(String[] args){
6          VehicleRoutingProblem.Builder vrpBuilder = VehicleRoutingProblem.Builder.newInstance();
7
8          Location n0 = Location.newInstance(3, -4);
9
10         Delivery n1 = Delivery.Builder.newInstance("n1")
11           .setLocation(Location.newInstance(-6, 8))
12           .addSizeDimension(UNITS_BEVERAGE, 15)
13           .setTimeWindow(TimeWindow.newInstance(30, 45))
14           .setServiceTime(3)
15           .build();
               .
               .
               .
62       VehicleType vehicleTypeF = VehicleTypeImpl.Builder.newInstance("unitVehicleTypeF")
63             .addCapacityDimension(UNITS_FOOD, 200).build(); // Vehicle food type definition
64       VehicleImpl vehicleInstanceF = VehicleImpl.Builder.newInstance("unitVehicleInstanceF")
65             .setType(vehicleType).setStartLocation(n0).build(); // Vehicle food instance defintion
66       vrpBuilder.addVehicle(vehicleInstanceF);// Adding vehicle food instance to the problem
67
68
69       VehicleType vehicleTypeB = VehicleTypeImpl.Builder.newInstance("unitVehicleTypeB")
70             .addCapacityDimension(UNITS_BEVERAGE, 150).build(); // Vehicle beverage type definition
71       VehicleImpl vehicleInstanceB = VehicleImpl.Builder.newInstance("unitVehicleInstanceB")
72             .setType(vehicleType).setStartLocation(n4).build(); // Vehicle food instance defintion
73       vrpBuilder.addVehicle(vehicleInstanceB); // Adding vehicle beverage instance to the problem
               .
               .
               .
62   }
63 }
```