## Design Entscheidungen:

### GUI:

Die GUI hat einige kleine Erweiterungen bekommen:

* Im Menü unter den Algorithmen finden sich nun die beiden neu implementierten Algorithmen Kruskal und Prim.

### Algorithmen:

Ähnlich wie bei den *ShortestPath* Algorithmen entschieden wir uns auch bei den *MinimalSpanningTree* Algorithmen dazu gegen ein Interface zu programmieren und somit die Algorithmen in der Verwendung leicht austauschbar zu machen.

Des Weiteren wurden auch die innerhalb der *MinimalSpanningTree* Algorithmen genutzten *EdgeFinder* und *VertexFinder* extrahiert, abstrahiert und hinter ein Interface gestellt.

Dies ermöglichte es uns auch innerhalb der Algorithmen sehr leicht zwischen der Verwendung dieser „Tools“ zu wechseln.

Somit gibt es auch keine Implementationsunterschiede innerhalb des Algorithmus zwischen den Versionen mit und ohne *FibonacciHeap*.

Sämtliche Unterschiede befinden sich hier in den extrahierten jeweiligen *EdgeFinder* Implementationen.

Unsere Implementation des *EdgeFinders* mit *FibonacciHeap* stützt sich auf die *FibonacciHeap* Implementation aus der JGraphT Bibliothek.

Die *VertexFinder* werden lediglich dazu verwendet um zu Beginn der Algorithmen einen Knoten auszuwählen, welcher dem Spannbaum hinzugefügt wird.

### Tools:

Um uns die Implementierung der Algorithmen einfacher zu gestalten haben wir erneut neue Tools erzeugt und alte Tools erweitert bzw. gekürzt, wenn ersichtlich wurde das einige Funktionen nicht benötigt waren.

Der *GraphenGenerator* wurde erweitert und nutzt nun eine neue Familie von Tools, nämlich die *HeuristikGeneratoren*, welche die Berechnung von Heuristiken zu übernehmen.

Außerdem gibt es nun ein Tool für Gewichtete Graphen welches einige häufig verwendete Funktionalitäten anbietet.

Entfernt wurden die Funktionalitäten des *GraphGenerators* Kreise und Räder zu erzeugen, da diese bereits in der JGraphT Bibliothek vorhanden sind.

## 2. Textantworten:

### 2. *Prioritätswarteschlangen* & *FibonacciHeaps*:

Bei einer Prioritätswarteschlange handelt es sich um eine Warteschlange welche nicht ein bloßes FIFO-Prinzip besitzt.

Stattdessen wird jedem Eintrag ein Wert zugeteilt und die Warteschlange anhand dieser Einträge sortiert.

Somit wird also nicht immer das zuerst eingefügte Element zurückgegeben, sondern jenes Objekt, welches die geringste Kennzahl besitzt.

2 Objekte mit gleicher Kennzahl werden jedoch normal nach FIFO behandelt.

Hierbei ist nicht gegeben, dass zu jedem Zeitpunkt alle Elemente in der Warteschlange an der richtigen Position sind.

Die richtige Ausgabereihenfolge kann ausschließlich beim wiederholten Ausgeben des Elements mit der kleinsten Kennzahl gewährleistet werden.

Hierfür gibt es verschiedene Implementationen, welche sich in ihrem Zeitaufwand stark unterscheiden können.

Eine dieser möglichen Implementationen benutzt *FibonacciHeaps*.

*FibonacciHeaps* bauen sich auf einer Collection aus Bäumen auf.

Hierbei gibt es gewisse Regeln zur Struktur dieser Bäume, welche dazu führen, dass die *FibonacciHeaps* einen konstanten Aufwand O(1) für das Auffinden des Minimums, das Einfügen neuer Einträge, das Verringern einer Kennzahl, des Verbinden zweier Bäume sowie einen Aufwand von O(log(n)) für das Entfernen des Eintrags mit der minimalen Kennzahl besitzen.

Somit sind sie äußerst effizient und nach derzeitigem Kenntnisstand mit eine der effizientesten Implementationsmöglichkeiten.

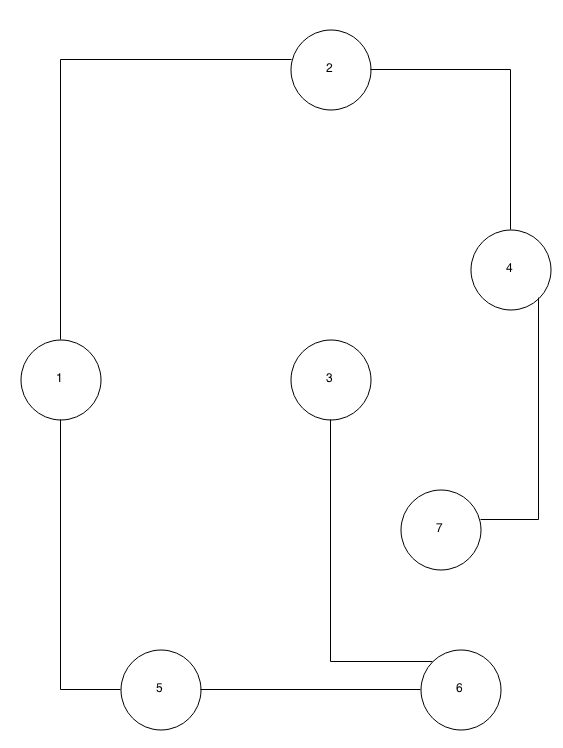
### 5. Laufzeit & Zugriffsverhalten von *MinimalSpanningTree* Implementationen:

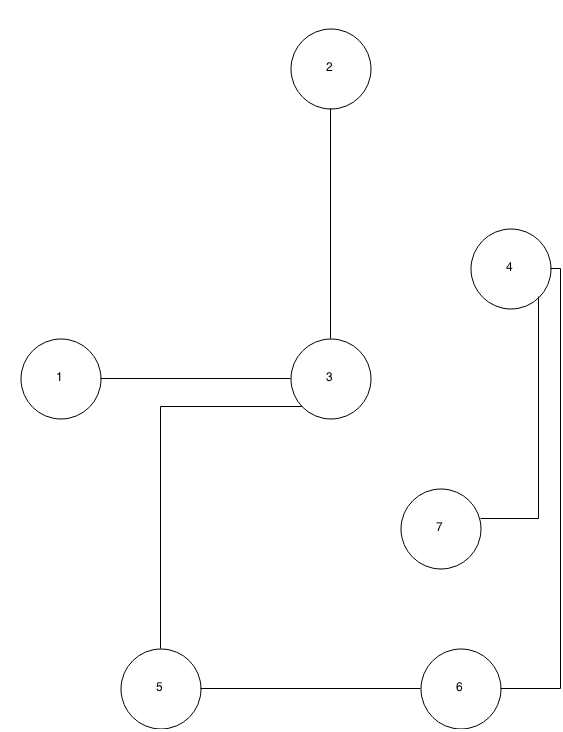
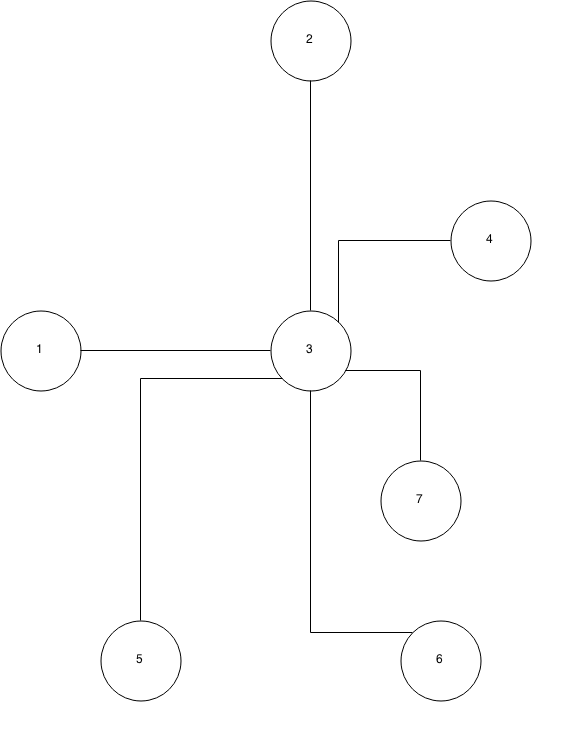
Lorem ipsum dolor sit amet, consectetur adipiscing elit.

## 3. Theorieteil:

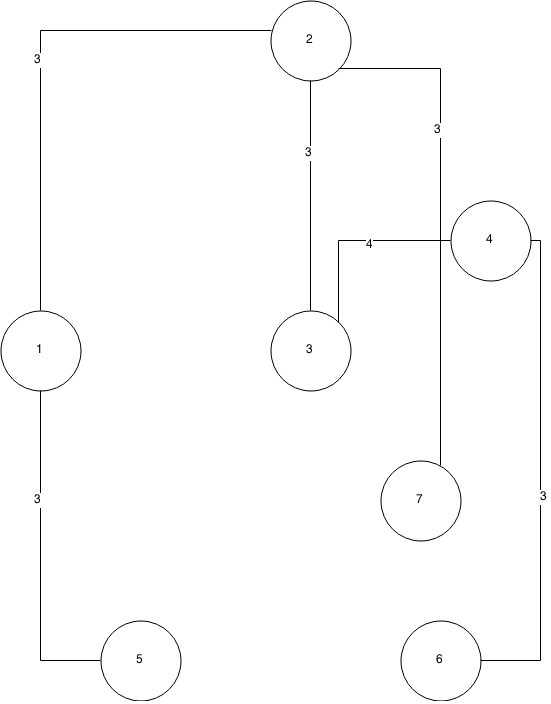
### Aufgabe VII:

1.)





2.)



### Aufgabe IX:

