Algodat Programmieraufgaben

Jakob Johannes Bauer 12002215

Inhalt

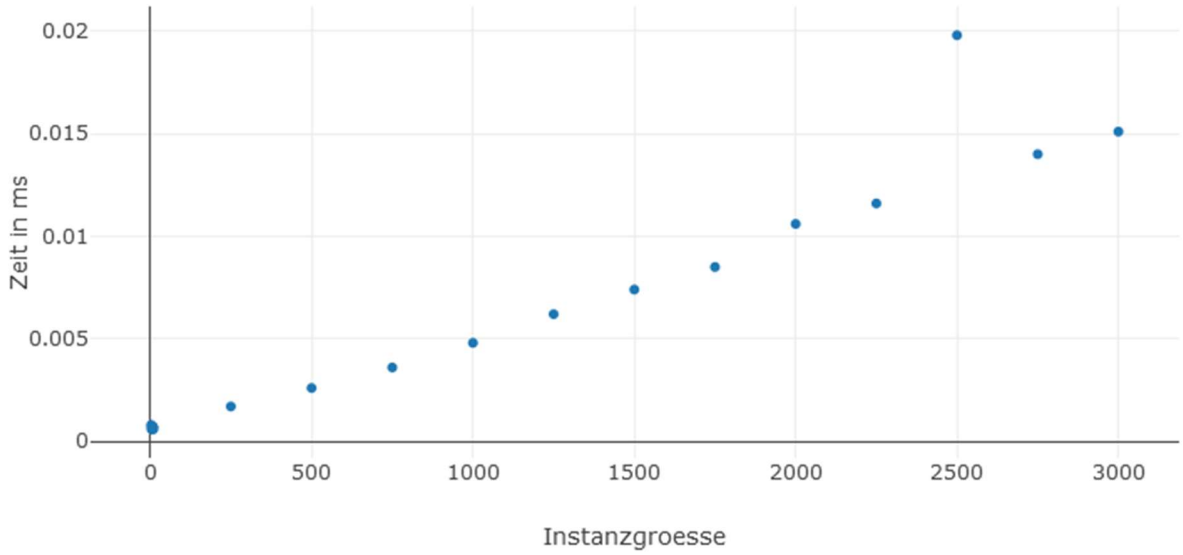
P1

Indexsuche

Die Indexsuche hat eine lineare Laufzeit von O(n) wie man am Plot erkennen kann.

Es existiert ein Ausreißer bei n=2500, wahrscheinlich wurde bei diesem n ein Testfall gewählt, bei dem die gesuchte Nummer nicht im System ist. Dadurch ist die Schleife komplett durchgelaufen und wurde nicht verfrüht abgebrochen.

Die Implementierung im Code ist, wie unschwer zu erkennen ist, von linearer Laufzeit, da genau eine Schleife mit der Obergrenze numbers.length existiert. Innerhalb der Schleife sind alle Operationen von konstanter Laufzeit, dadurch ergibt sich O(n) was sich mit den Beobachtungen des Plots deckt



Gale Shapley

Der Gale Shapley hat eine Form von exponentieller Laufzeit.

Da bei n=2000 die Zeit 13.41ms beträgt, und bei n=3000 33.73ms, kann der Graph näherungsweise durch eine Funktion zweiten Grades beschrieben werden.

Dadurch ergibt sich eine Laufzeit von O(n^2)

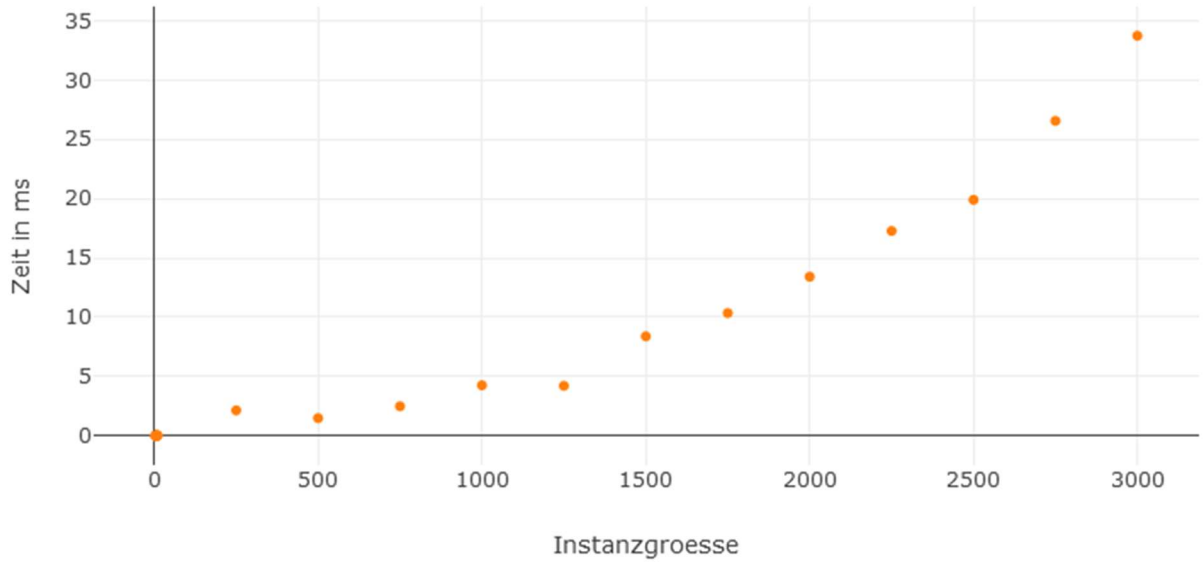
Es könnte sich auch um eine Funktion mit O(x^n) mit x element aus R, jedoch bräuchte man größere Messwerte, um dies bestimmen zu können. Dadurch, dass der Graph nicht extrem rasant ansteigt, vermute ich O(n^2).

Aus der Vorlesung wissen wir bereits, dass der Gale Shapley Algorithmus eine Laufzeit von O(n^2) aufweist. Wenn wir den Code näher analysieren kann man aufschlüsseln, wie dieses Ergebnis zu Stande kommt.

Die while Schleife hat eine Laufzeit von O(n^2), da es maximal n^2 mögliche Verbindungen zwischen den beiden Parteien gibt.

Innerhalb der while Schleife gibt es noch eine for Schleife zur Selektion der ersten freien Familie, da diese jedoch auch auf die Verbindungen zwischen den Parteien zugreift, und wir vorhin gezeigt haben, dass es nur n^2 Verbindungen gibt, ist die Gesamtlaufzeit O(n^2).

Alle anderen Operationen haben Konstante Laufzeit, bis auf das anfängliche Befreien der Verbindungen, was jedoch O(n^2 + n) ergibt, was das selbe wie O(n^2) ist.



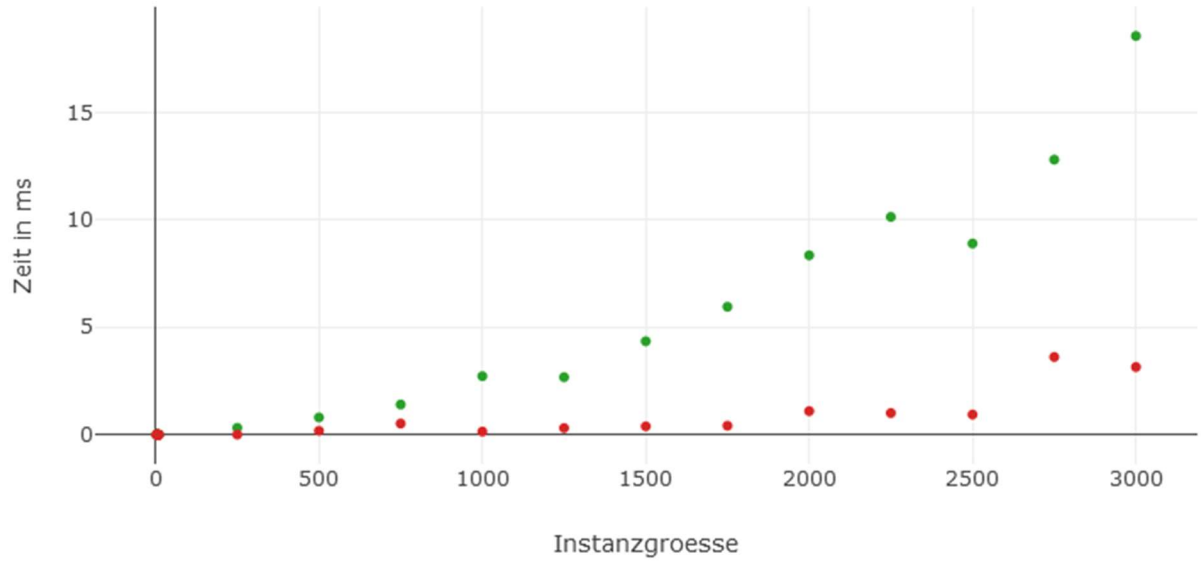
Stable Matching Positive & Negative

Auch bei der Überprüfung des Gale Shapley Algorithmus ist zu erkennen, dass es sich um O(n^2) handelt.

Bei genauerer Betrachtung des Java Codes sieht man leicht, dass es sich um zwei verschachtelte Schleifen handelt, die jeweils n als obere Grenze haben. Die Operationen innerhalb der inneren Schleife sind alle Konstant, wodurch sich direkt eine Laufzeit von O(n^2) ergibt.

Wenn man sich die negativen Testfälle anschaut, sind zwei Dinge zu beobachten:

1. Die Laufzeit erfolgt immer schneller als im positiven Testfall, dies liegt daran, dass sie dieselbe obere Schranke haben, jedoch immer schneller fertig sein müssen, da sie die Schleifen nie komplett durchlaufen
2. Die Laufzeit inkonsistent ist.
   Dadurch, dass die gesamte Operation abgebrochen wird, sobald ein unstabiles Paar gefunden wurde, ist die reale Laufzeit nicht nur von n abhängig. Dadurch kann man keine Regelmäßigkeit anhand der n-Achse feststellen



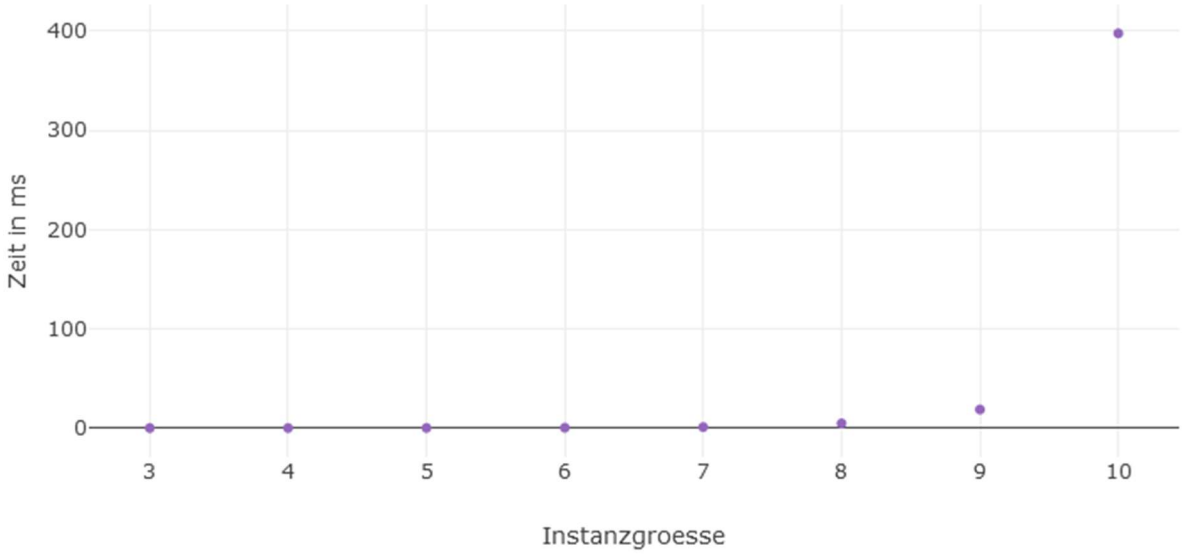
Brute Force

Anhand des Plots erkennt man aufgrund des rasanten Anstiegs im Bereich n gleich 9 bis 10 eine exponentielle Laufzeit, die sehr stark ansteigt.

Für n = 11 ist anzunehmen, dass die Steigerung von n=10 auf n=11 größer ist als die Steigerung von n=9 auf n=10.

Das Verhältnis der Steigerung von n=8 bis n=9 zu n=9 bis n=10 ist etwa (397-18)/(18 – 4), also fast 32 mal. Wenn mit diesem Verhältnis fortgesetzt werden würde, wäre die Laufzeit bei n= 11:

397ms * 32 = 12 700, also fast 13 Sekunden.



P2

Dijkstra und A*

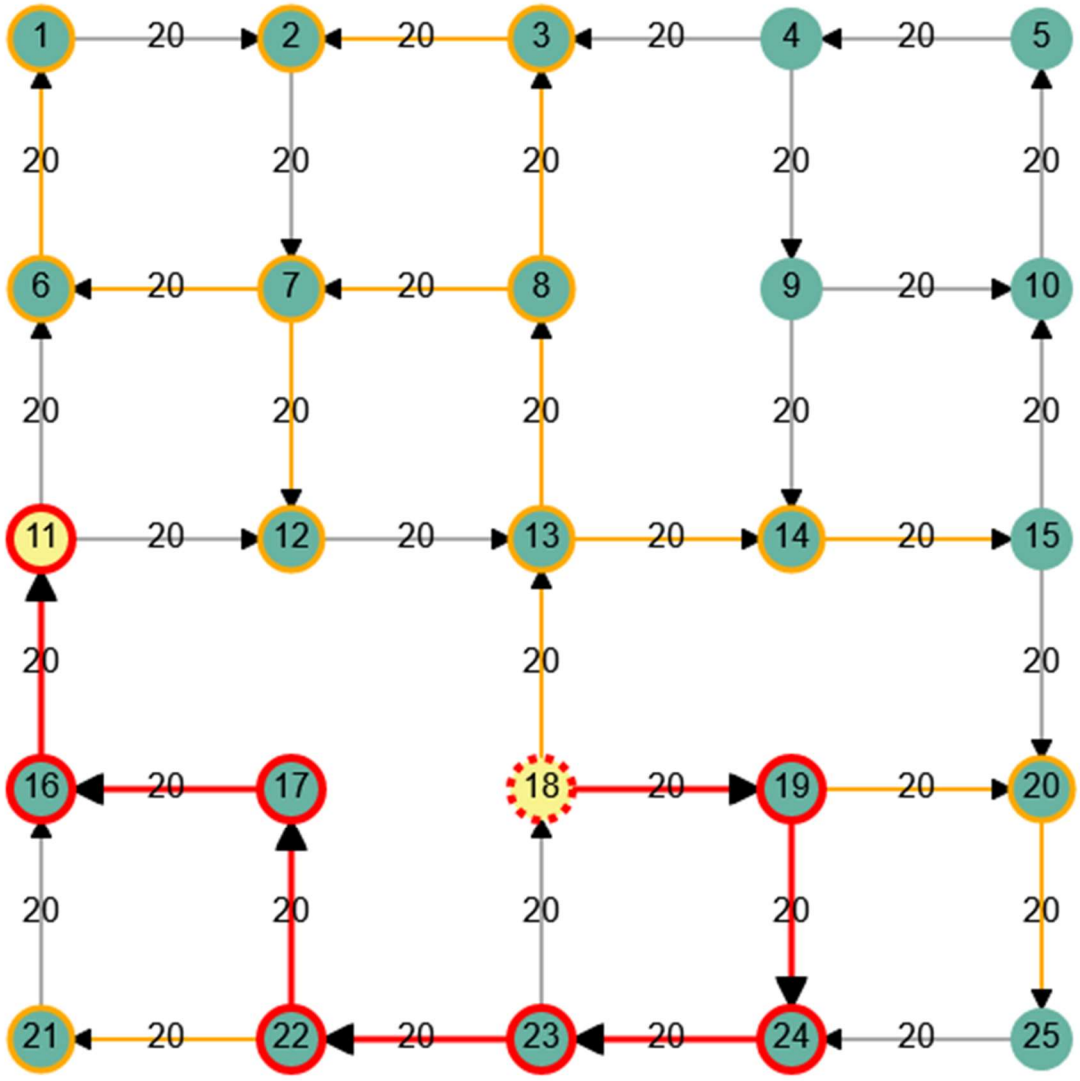
Der A* Algorithmus ist dem Dijkstra Algorithmus genau dann äquivalent, wenn die Null-Heuristik verwendet wird, da genau dies den Unterschied zwischen den zwei Algorithmen ausmacht. Der Dijkstra Algorithmus hat keine Schätzung von bekannten Nodes zum Ziel, was äquivalent dazu ist, alle Kanten nur mit dem Gewicht zu berechnen.

Dijkstra erforscht also alle möglichen Pfade, während A* versucht, die besten Chancen zu priorisieren.

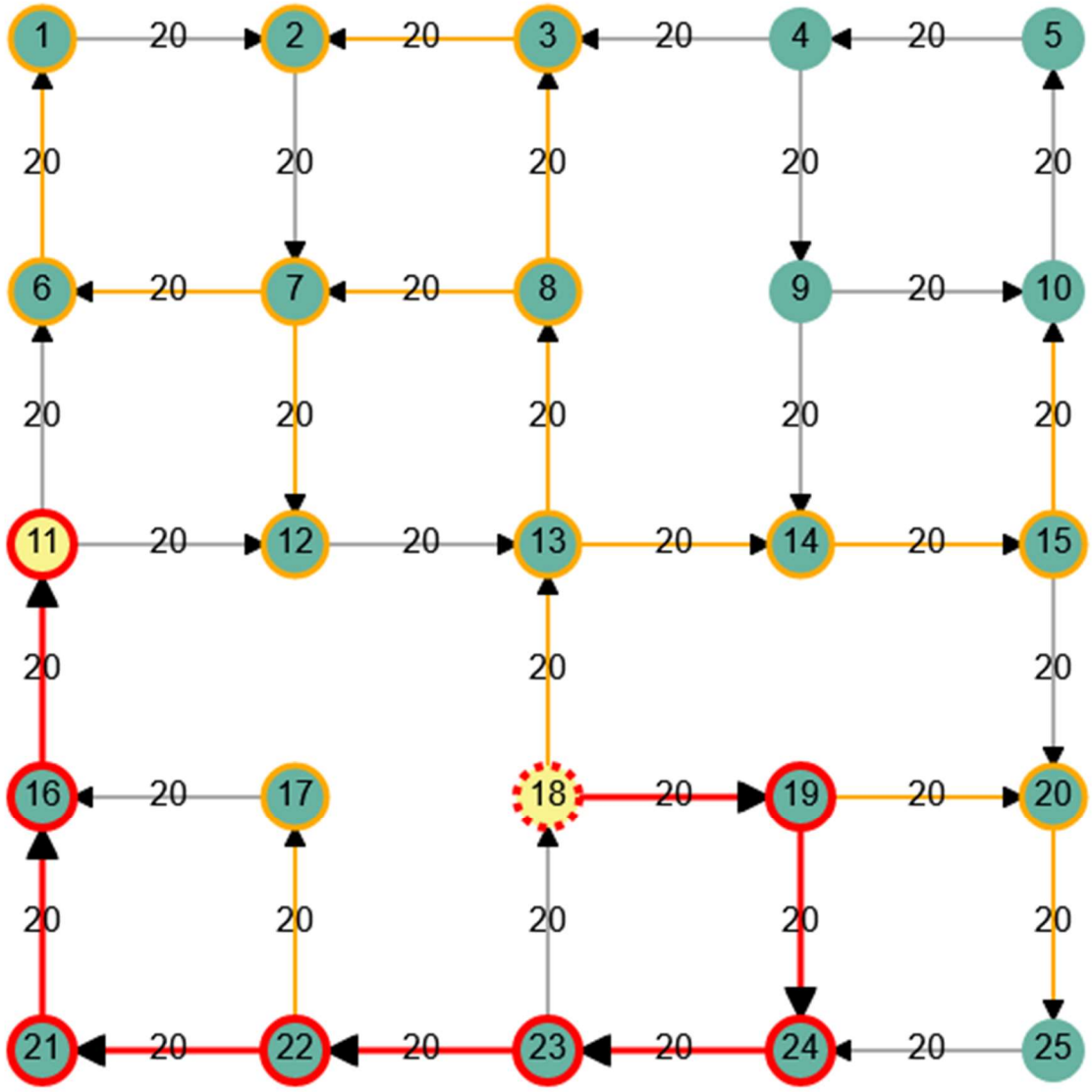
Diese Heuristik ist zulässig, da 0 immer geringer als die minimalen Kosten zwischen zwei oder mehr Knoten ist.

Instanz 540 vs. Instanz 1165

Instanz 540 – Euklidische Distanz



Instanz 1165 – Manhatten Distanz



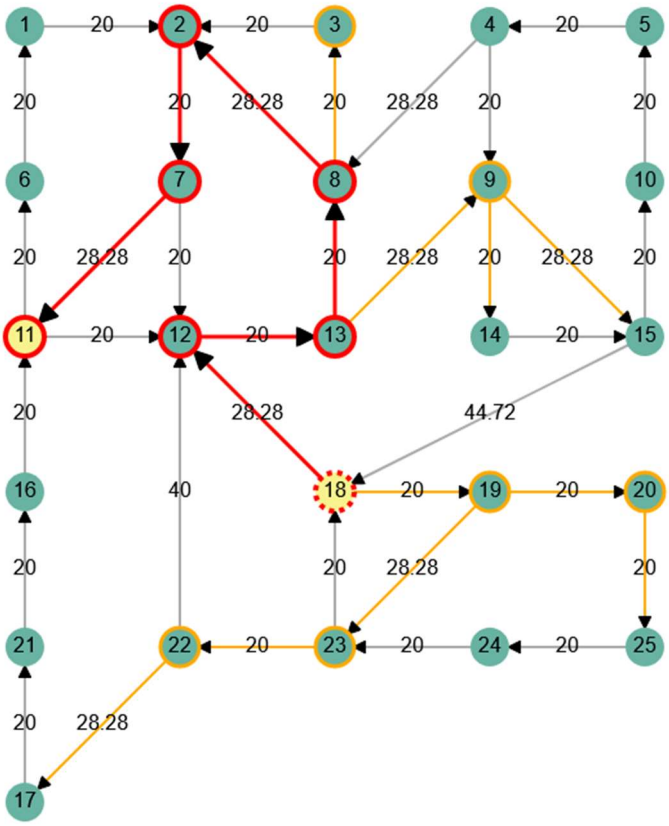
Vergleich

Beide Distanzen ergeben einen sehr ähnlichen Pfad, mit der Ausnahme der Nodes 17 und 21, welche beim jeweils anderen Pfad in-/exkludiert werden. Da die Wegkosten jedoch zwischen allen Nodes gleich sind, sind beide Lösungen gleichwertig.

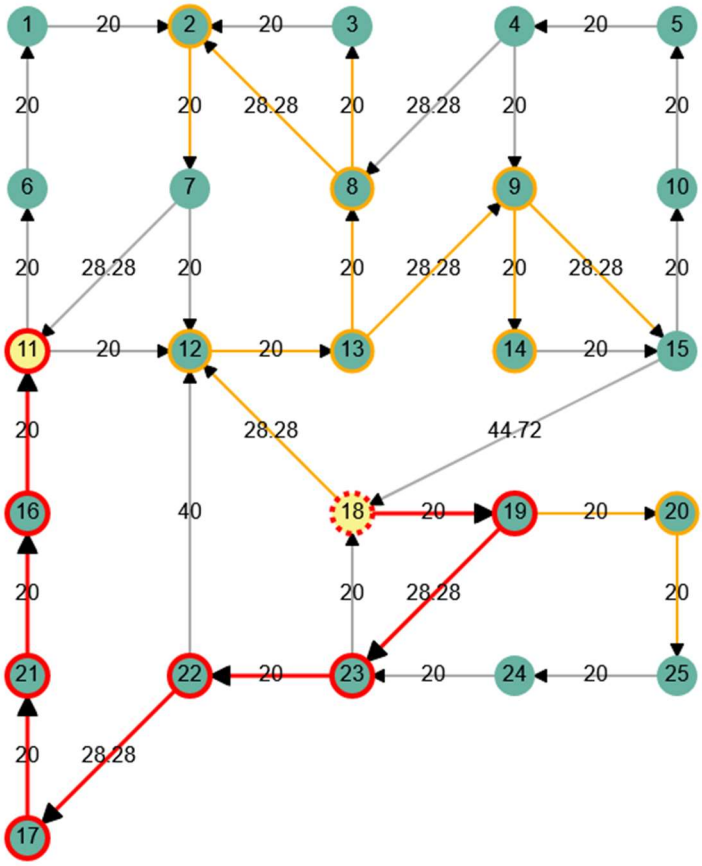
Daraus ergibt sich auch, dass die Heuristiken zulässig sind, da es keinen besseren Weg gibt. Wahrscheinlich ergeben sich Unterschiede zwischen den verschiedenen Heuristiken, wenn die Wegkosten zwischen den Nodes nicht konstant sind. In diesem konkreten Experiment sind sie jedenfalls gültig.

Instanz 3040 vs. Instanz 3665

Instanz 3040 – Euklidische Distanz



Instanz 3665 – Manhatten Distanz



Vergleich

Diesmal ist zwischen den beiden Heuristiken ein Unterschied im Pfad zu sehen, was die Theorie der Abhängigkeit von konstanten Wegkosten fördert.

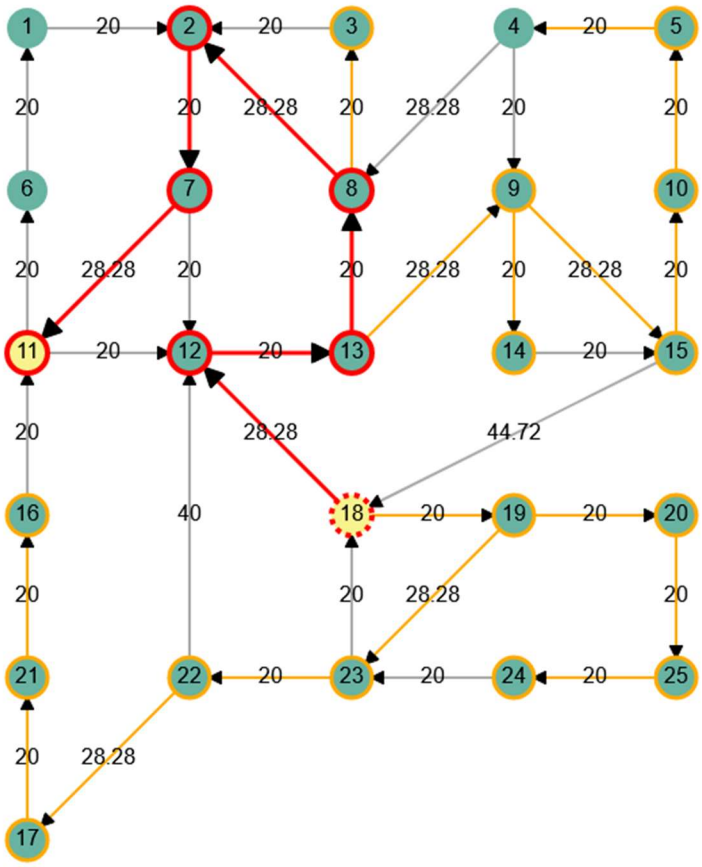
Im direkten Vergleich ergeben sich für den gefundenen Pfad in der Euklidischen Metrik Wegkosten von 144, in der Manhatten Distanz aber 156. Demnach ist die Manhatten Distanz in diesem Graphen keine zulässige Heuristik, die Euklidische Distanz kann nach eigenem Trial-and-Error als korrekt eingestuft werden.

Instanz 3040 vs. Instanz 4915

Instanz 3040 – Euklidische Metrik

Hierzu siehe die bereits eingefügte Grafik weiter oben

Instanz 4915 – Null-Heuristik



Vergleich

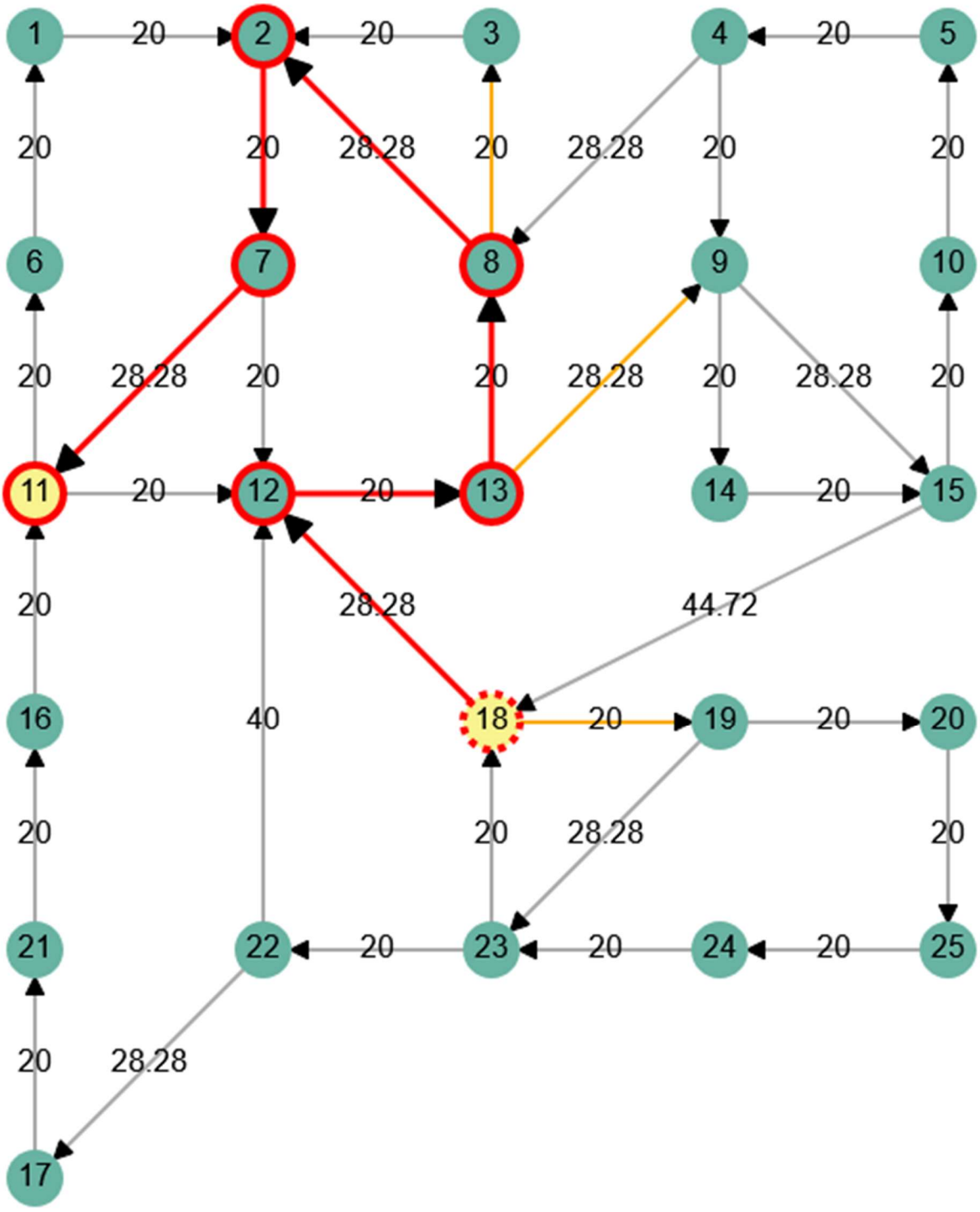
Beide Grafen finden dasselbe, günstigste Endresultat, es sind also beide Heuristiken Lösungen. Das ist auch in der Theorie unschwer zu erkennen, da weiter oben bereits argumentiert wurde, dass ein A* Algorithmus, welcher die Null-Heuristik verwenden, ein Dijkstra Algorithmus ist. Dieser findet immer den kürzesten Weg, da jeden Pfad ausprobiert, solange er nicht unzulässig ist. Das ist auch daran zu erkennen, dass die Null-Heuristik nicht so effizient war (23 Schritte statt 14).

Instanz 3040 vs. Instanz 4290

Instanz 3040 – Euklidische Metrik

Hierzu siehe die bereits eingefügte Grafik weiter oben

Instanz 4290 – Shortest Path



Vergleich

Beide Heuristiken liefern das richtige Ergebnis zurück, wobei Shortest Path nur 8 statt 14 Schritte benötigt. Dies ist trivial, da der kürzeste Pfad bei Shortest Path bereits bekannt ist, und dieser Weg 7 Nodes beinhaltet.

Demnach sind beide Heuristiken zulässig, Shortest Path kann aber nur verwendet werden, wenn der kürzeste Weg bereits im Vorhinein ermittelt und gespeichert wurde, wie es Beispielsweise bei Google Maps der Fall ist. Zur Entdeckung/Berechnung von neuen Wegen, macht dies jedoch keinen Sinn.

P3

Funktionsweise

Anfangs wurde der BinaryTree Algorithmus rekursiv implementiert. Dabei wird eine Node beim einfügen mit dem Key der derzeitigen Node verglichen. Hier kann es zu drei unterschiedlichen Zuständen kommen:

1. Die Nodes haben den gleichen Key. In diesem Fall soll die Node einfach ausgelassen werden (return ohne Änderung)
2. Die neue Node hat einen größeren Key. Nun soll die neue Node als rechtes Child der alten Node eingefügt werden. Wenn dieses rechte Child bereits existiert, wird stattdessen die selbe Methode auf dieses Child angewandt. So kann die neue Node Schritt für Schritt ihrem Ziel nähergebracht werden, bis sie tatsächlich eingefügt wird.
3. Das selbe wie Schritt Zwei, nur als left Child

Diese Implementierung funktionierte tadellos, jedoch stürzt sie im Test aufgrund eines Stackoverflow Errors ab. Dies liegt daran, dass die Rücksprungadresse bei jedem Methodenaufruf im Stack gespeichert wird. Wenn es zu viele verschachtelte Methodenaufrufe gibt, und die Sprache die verwendet wird keine besondere Funktionalität diesbezüglich unterstützt, läuft der Stack zu voll.

Es ist also festzuhalten, dass für besonders tiefe Bäume, eine rekursive Implementation nicht immer funktioniert.

Um dieses Problem zu lösen, wurde die Methode iterativ implementiert. Dazu wurde der Code kopiert, in eine while Schleife eingefügt, und eine bufferNode eingerichtet, welche statt eines rekursiven Aufrufs der Methode so aktualisiert wird, dass sie immer die Adresse des jeweiligen Childs von sich selbst enthält.

Der AVL-Tree fügt jedes Mal nach diesem Einfügen noch einen Schritt hinzu, nämlich die Rebalancierung, falls diese notwendig ist. Dabei wird verglichen, ob die Höhendifferenz zwischen der zwischen der eingefügten Node und der linked childNode der parentNode (falls diese existiert) 2 ist. Da in einem AVL-Tree die Differenz in der Höhe maximal 1 sein darf, muss dieser Baum nun rebalanciert werden. Dies wird umgesetzt, indem zwischen zwei Fällen unterschieden wird:

1. Wenn das zweifach rechte Child von p gleichhoch oder höher wie das rechts-linke Child von p ist, muss p einmal nach links gedreht werden. So wird die Höhe des Teilbaumes um 1 reduziert, aber die logische Sortierung ist noch immer korrekt, da das neue leftChild kleiner ist als das neue p, welches kleiner ist als das neue rightChild (Fall 2.1 der VO)
2. Andernfalls ist der der Teiltree des rechten Childs einmal nach rechts zu rotieren, und anschließend der gesamte Teiltree nach links (Fall 2.2 der VO)

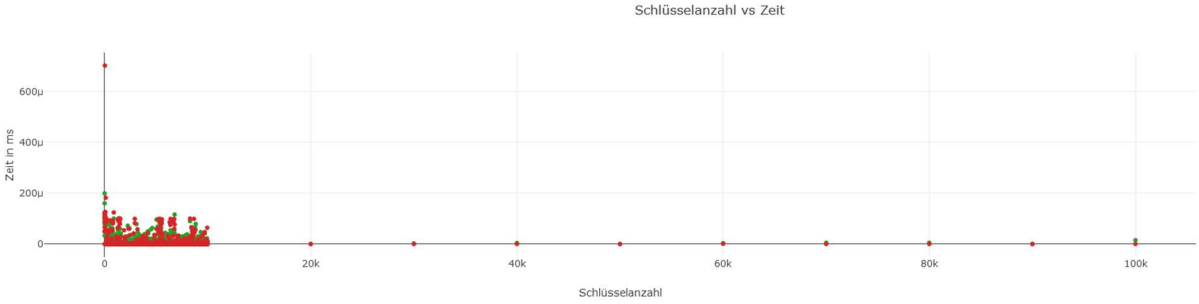
Analog passiert es, für die rechten Children.

Rekursive Aufrufe – Binary Tree vs AVL Tree

Im Testdurchlauf ergibt sich, dass diese Methode rekursiv implementiert werden kann. Das liegt daran, dass durch die Rebalancierung der Höhenunterschied maximal 1 beträgt, der Baum also besonders breit gefächert ist. Dadurch wird er logischerweise weniger tief, und es werden nicht so viele rekursive Aufrufe benötigt um diesen abzurechnen. Jedoch wird dieses Problem theoretisch wieder auftreten, sobald sehr viel mehr Nodes hinzugefügt werden. In der Praxis ist das jedoch nicht der Fall, da der neue Worstcase der rekursiven Aufrufe nichtmehr n ist (mit n = Anzahl nodes), sondern ld(n) (maximaleNodes = 2^Baumtiefe)

Für den binary Tree ergibt sich der Worstcase genau dann, wenn die Eingabe in (aufsteigend oder absteigend) sortierter Reihenfolge erfolgt, da jede Ebene dann genau eine Node enthält, und somit die maximale Anzahl notwendiger rekursiver Aufrufe erreicht wird.

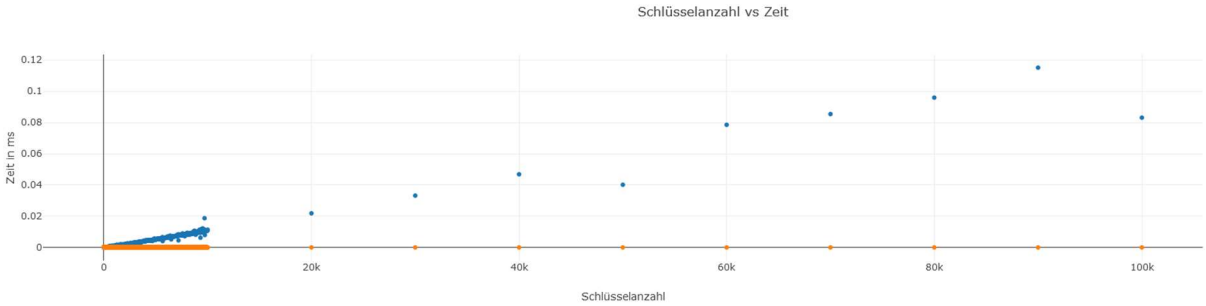
Vergleich der Suchlaufzeiten – Shuffled Input



Durch einen Blick auf den Graphen erkennt man, dass es keinen sichtlichen Unterschied zwischen den Suchlaufzeiten im Binary Tree und im AVL Tree gibt. Vor allem bei kleineren Werten sind die Suchzeiten sehr zufällig und stark verteilt, bei großen Werten erkennt man eine erhöhte Regelmäßigkeit. Dies liegt daran, dass Wahrscheinlichkeiten und Statistiken bei großen Mengen gelten, sich auf kleine Stichproben jedoch kaum auswirken, und mehr Suchen für kleine Werte durchgeführt wurden.

Der Grund, warum der Unterschied hier so klein ist, ist, dass bei einem zufälligen Input, die Daten auch im Binary Tree sehr regelmäßig verteilt sind. Dadurch erhalten wir einen breiteren und flacheren Baum, was weniger Aufrufe zum Suchen benötigt.

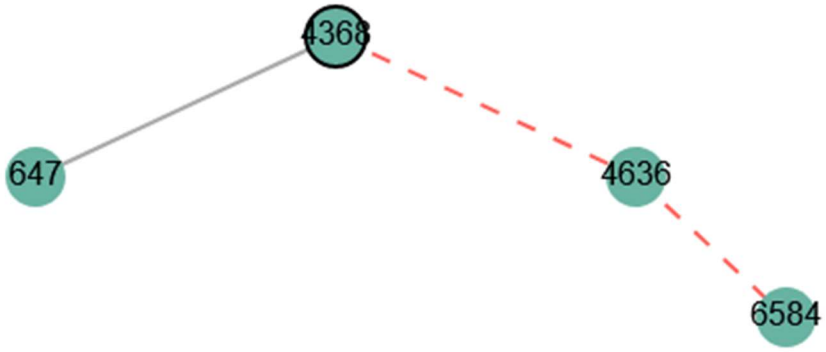
Vergleich der Suchlaufzeiten – ordered Input



Hier erkennt man sofort eine klare Diskrepanz zwischen den beiden Baumstrukturen. Die Suche in der AVL Tree ist um einiges schneller, da die Schleife/Rekursion durchschnittlich nicht so oft aufgerufen werden muss, um ans Ziel zu gelangen. Die Argumentation hierfür ist Analog zu Argumentation unter „Rekursive Aufrufe“

Direkter Vergleich der Struktur

Binary Search Tree

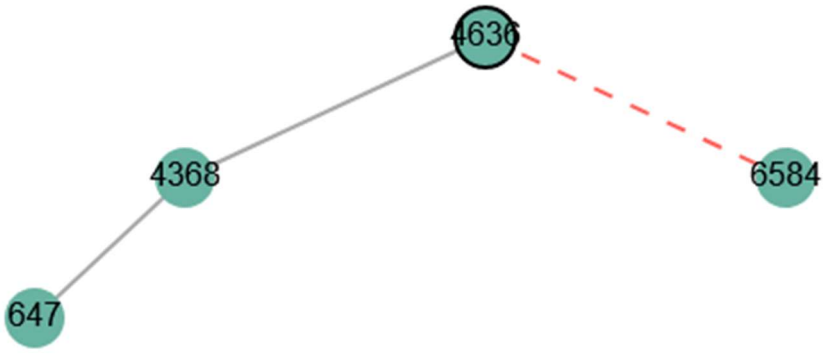


Dieser Tree hier ist balanciert, da sich die Anzahl der rechten und linken Knoten für jeden Knoten maximal um 1 unterscheiden. Bezüglich der Reihenfolge des Einfügens kann man einige Aussagen tätigen, eine komplette Widerherstellung des Prozesses ist jedoch nicht möglich.

Alle möglichen Reihenfolgen die sich aus dieser Visualisierung ergeben sind:

- 4368, 647, 4636, 6584
- 4368, 4636, 547, 6584
- 4368, 4636, 6584, 647

AVL Tree



Der AVL Tree ist ebenso balanciert, hier kann jedoch kaum eine Aussage über die Einfügereihenfolge getätigt werden, da sich durch die Rotation alles verschieben kann. Da wir jedoch aus dem oberen Binary Search Tree bereits nur drei Möglichkeiten haben, können wir eventuell eine oder mehrere ausschließen.

- 4368, 647, 4636, 6584
  Angenommen, diese Reihenfolge würde im AVL Tree eingefügt worden sein, dann hätte zu keinem Zeitpunkt rebalanced werden müssen. Demnach müssten die beiden Trees äquivalent sein. Da sie das nicht sind, ist diese Reihenfolge nicht die richtige.

- 4368, 4636, 547, 6584
  Auch hier müsste der AVL Tree nie rebalanced werden. -> Diese Reihenfolge ist inkorrekt

- 4368, 4636, 6584, 647
  Durch das Ausschlussprinzip ergibt sich bereits, dass es sich hier um die richtige Reihenfolge handelt. Dies werden wir nun kontrollieren

Add 4368 -> Head = 4368

Add 4636 -> Head = 4368, RightChild = 4636

Add 6584 -> Head = 4368, RightChild = 4636, RightRightChild = 6584

Nun ist der Tree unbalanced, Head ist kleiner als RightChild -> Leftrotation

Head = 4636, LeftChild = 4368, RightChild = 6584

Add 647 -> Es ergibt sich die gegebene Datenstruktur

Sequenz für idente Bäume

Ein einfaches Verfahren nach dem man vorgehen kann um die beiden Datenstrukturen ident zu halten ist, je eine Ebene von links nach rechts zu befüllen, bis diese voll ist. Damit dies passiert, müssen die Keys passend gewählt werden.

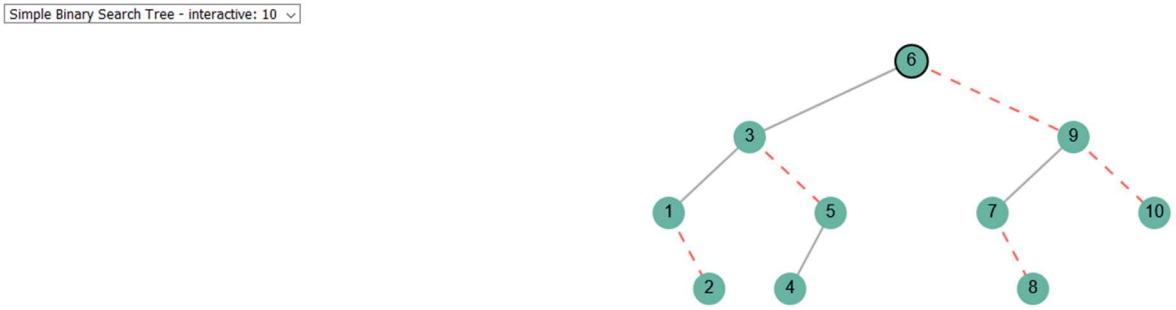
Ich nehme in diesem Beispiel der Übersichtlichkeit halber die Keys 1 bis 10. Dabei beginne ich mit dem mittleren, dann den mittleren der unteren Hälfte, den mittleren der oberen Hälfte, den mittleren des ersten Viertels, des zweiten Viertels usw. Gegebenenfalls Runde ich auf/ab

Somit ergibt sich folgende Sequenz:

6, 3, 9, 1, 5, 7, 10, 2, 4, 8

Daraus entstehen diese Bäume:

Binary Search Tree:



AVL Tree: