NAVIGATION PROJECT

A trip through Switzerland (Basel-Allschwil)

Herbstsemester 2025/2026 - FHNW

Kapischan Sriganthan & Mladen Radovanovic

Management Summary

Dieses Projekt befasst sich mit der Entwicklung, Implementierung und Analyse verschiedener Suchalgorithmen zur Wegfindung auf einer eigens erstellten Kartendatenbasis im Raum Allschwil – Basel.

In der ersten Phase lag der Fokus auf der Datenerhebung und -aufbereitung, um eine funktionsfähige Karte zu erstellen. Hierzu wurden sämtliche relevanten Knoten (Nodes) und Kanten (Edges) manuell erfasst und zu einer strukturierten Datengrundlage zusammengeführt.

Die zweite Phase widmete sich dem Vergleich klassischer Suchverfahren. Dazu wurden die Algorithmen Depth-First Search (DFS), Breadth-First Search (BFS), Best-First Search und A* implementiert, getestet und hinsichtlich ihrer Funktionsweise, Laufzeit sowie Ergebnisqualität systematisch miteinander verglichen.

In der dritten Phase wurde das Projekt um weitere, neue Verfahren erweitert, darunter der Dijkstra-Algorithmus und die Beam Search. Diese wurden ebenfalls implementiert und auf derselben Kartendatenbasis analysiert. Zusätzlich wurde der A*-Algorithmus durch die Integration einer neuen Heuristik („Speed Limits“) optimiert, um die Wegsuche noch realitätsnäher und effizienter zu gestalten.

Das gesamte Projekt – einschliesslich des vollständigen Quellcodes, der Datensätze sowie der Dokumentation – ist im GitLab-Repository unter folgendem Link abrufbar:
https://gitlab.fhnw.ch/mladen.radovanovic/SoftwareEngineeringProjects.git

Inhaltsverzeichnis

Management Summary 1

Phase 1 – Kartendaten erstellen 3

Phase 2: Testen vorhandener Algorithmen 5

- Depth-First Search 5
  - Funktionsweise 5
  - Resultat 6
- Breadth-First Search 7
  - Resultat 8
- Best-First Search 9
  - Funktionsweise 9
  - Resultat 10
- A* Search 11
  - Funktionsweise 11
  - Resultat 13

Phase 3: Neuer oder angepasster Algorithmus 14

- Dijkstra-Algorithmus 14
  - Funktionsweise 14
  - Resultat 15
- Beam Search 16
  - Funktionsweise 16
  - Resultat 17
- A* mit neuer Heuristik «Speed Limit» 18
  - Resultat 19

Vergleich und Bewertung der Algorithmen 20

Phase 1 – Kartendaten erstellen

Ziel der ersten Phase war der Aufbau einer eigenen Kartendatenbasis für einen Teil des Raums Allschwil–Basel. Dazu wurden relevante Kreuzungen (Nodes) und Strassen (Edges) manuell erfasst und in zwei Datendateien gespeichert:

- Datei nodes.txt: Name;Koordinate East;Koordinate North
- Datei edges.txt: StartNode;EndNode;Distanz

Die Datengrundlage bildeten Karten von map.geo.admin.ch, wodurch eine realistische, aber kompakte Testumgebung entstand.

(Blau = Kapischan, Rot = Mladen, Grün = Verbindungsstrassen).

Zur Datenprüfung wurde die Klasse MapData implementiert. Sie liest beide Dateien ein und erzeugt:

- **HashMap** mit Nodes und ihren Koordinaten (Map<String, GPS>)
- **Adjacency-Liste** mit allen Verbindungen (Map<String, ArrayList<Destination>>).

Ein Testcode überprüft den Import, zählt die Nodes und Edges und gibt die Daten aus:

```
MapData mapData = new MapData();
```

```
Map<String, MapData.GPS> nodes = mapData.getNodes();
int sumNode = 0;
System.out.println("Alle Nodes mit den Koordinaten: \n".toUpperCase());
for (String node : nodes.keySet()) {
    MapData.GPS gps = nodes.get(node);
    sumNode++;
    System.out.println(node + ": (" + gps.north() + "," + gps.east() + ")");
}
System.out.println("============================================================");
System.out.println("ANZAHL NODES: " + sumNode);
System.out.println("============================================================");
```

```java
Map<String, ArrayList<MapData.Destination>>edges = mapData.getAdjacencyList();

int sumEdges = 0;
System.out.println("Alle Edges mit den Distanzen: \n".toUpperCase());
for(String node: edges.keySet()) {
    for(MapData.Destination neighbour : edges.get(node)) {
        System.out.println(node + " -> " + neighbour.node() + " (" + neighbour.distance() +
                "m / Speed Limit: " + neighbour.speedLimit() + ")");
        sumEdges++;
    }
}
System.out.println("============================================================");
System.out.println("ANZAHL Edges: " + sumEdges);
System.out.println("============================================================");
```

Beispielhafte Konsolenausgabe:

```
ALLE NODES MIT DEN KOORDINATEN:

BaselAllschwilerstrasse9: (1267216.0,2610320.0)
BaselAllschwilerstrasse8: (1267236.0,2610265.0)
AllschwilSpitzwaldstrasse8: (1267357.0,2609245.0)
AllschwilSpitzwaldstrasse6: (1267276.0,2609175.0)
AllschwilSpitzwaldstrasse7: (1267316.0,2609207.0)
```

```
ALLE EDGES MIT DEN DISTANZEN:

BaselAllschwilerstrasse9 -> BaselAllschwilerstrasse8 (59.2m / Speed Limit: 50)
BaselAllschwilerstrasse9 -> BaselSpalenring1 (102.8m / Speed Limit: 50)
BaselAllschwilerstrasse8 -> BaselAllschwilerstrasse7 (82.9m / Speed Limit: 50)
BaselAllschwilerstrasse8 -> BaselAllschwilerstrasse9 (59.2m / Speed Limit: 50)
```

Nach der Erhebung standen **73 Nodes** und **224 Edges** zur Verfügung.

(Speed Limit wurde in Phase 3 ergänzt.)

Phase 2: Testen vorhandener Algorithmen

In dieser Phase wurden verschiedene Suchverfahren implementiert und auf der Kartendatenbasis aus Phase 1 getestet. Ziel war der Vergleich von Funktionsweise, Laufzeit und Ergebnisqualität.

Depth-First Search

DFS verfolgt einen Pfad **tiefst möglich**, springt bei Sackgassen zurück und prüft Alternativen.

Funktionsweise

Start- und Zielknoten werden an die Hauptmethode übergeben → **depthFirst** liefert Pfad als Resultat:

```
private static final String start = "AllschwilBaslerstrasse1";  2 usages
private static final String end = "BaselSpalenring1";  2 usages
```

```
ArrayList<String> path = depthFirst(start, end);
```

Zunächst wird der Startpunkt hinzugefügt, anschliessend erfolgt die Suche rekursiv.

```
private static ArrayList<String> depthFirst(String start, String end) {  1 usage   mladen.radovanovic +1
    ArrayList<String> path = new ArrayList<>();
    path.add(start);
    return depthFirstRecursive(path, start, end);
}
```

Die rekursive Methode erhält den aktuellen Pfad, die Strasse und das Ziel. Ist das Ziel erreicht, wird der Pfad zurückgegeben. Andernfalls werden die Nachbarn des aktuellen Knotens ermittelt und geprüft, ob eine Sackgasse vorliegt:

```
private static ArrayList<String> depthFirstRecursive(ArrayList<String> path, String current, String end) {
    // Jeder Aufruf = ein geprüfter Pfad (Expansion dieses Knotens)
    pathsChecked++;

    // Basisfall: Ziel erreicht -> aktuellen Pfad zurückgeben
    if (current.equals(end)) {
        return new ArrayList<>(path);
    }

    ArrayList<MapData.Destination> neighbors = adjList.get(current);
    if (neighbors == null) {
        return null; // Sackgasse -> kein Pfad über diesen Ast
    }
```

In einer Schleife wird ein Nachbar nach dem anderen geprüft. Befindet sich ein Knoten bereits im Pfad, würde ein Zyklus entstehen. In diesem Fall wird die aktuelle Iteration übersprungen und mit dem nächsten Nachbarn fortgefahren.

Anschliessend wird der Nachbar zum Pfad hinzugefügt, und die Methode ruft sich selbst auf, um der Logik weiter in die Tiefe zu folgen.

Liefert der rekursive Aufruf ein Ergebnis, wurde ein vollständiger Pfad zum Ziel gefunden und sofort zurückgegeben. Führt er zu keiner Lösung, wird der Nachbar entfernt und die Schleife mit dem nächsten Knoten fortgesetzt. So werden systematisch alle möglichen Wege geprüft, bis ein Ziel erreicht oder alle Optionen ausgeschöpft sind:

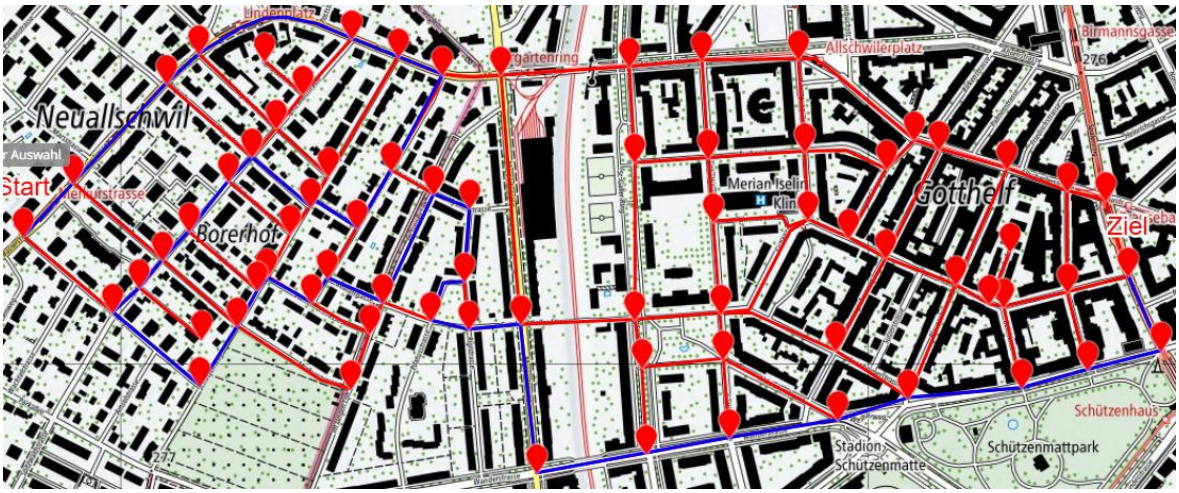
```java
for (MapData.Destination dest : neighbors) {
    String next = dest.node();
    // Cycle vermeiden: Knoten schon im aktuellen Pfad?
    if (path.contains(next)) {
        rejectedCycles++;
        continue;
    }
    // Knoten betreten
    path.add(next);
    neighborsVisited++;

    ArrayList<String> result = depthFirstRecursive(path, next, end);
    if (result != null) {
        return result; // Lösung gefunden -> direkt nach oben durchreichen
    }
    // Backtracking
    path.remove( index: path.size() - 1);
}
// Keine Lösung in diesem Teilbaum
return null;
```

Hinweis: Methoden wie computePathDistance im Code dienen lediglich der statistischen Auswertung; eine vertiefte Analyse erfolgt hier und bei allen folgenden Algorithmen nicht.

Resultat

Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
========================================
Status: Depth-First-Search funktioniert wie erwartet!
--------------------------------------------------------
Benötigte Zeit: 0.342 Millisekunden
Pfadlänge: 38
Distanz: 3608.88 Meter
Anzahl geprüfter Pfade: 42
Anzahl Nachbarn besucht: 41
Abgelehnte (wegen Cycle-Gefahr): 36
--------------------------------------------------------
```



Breadth-First Search

Breadth-First Search durchsucht den Graphen schichtweise (erst alle Wege mit 1 Kante, dann 2, usw.) und liefert damit den kürzesten Pfad nach Kantenanzahl zwischen start und end.

Start- und Zielknoten werden an die Hauptmethode übergeben → **breadthFirst** liefert Pfad als Resultat:

```
private static final String start = "AllschwilBaslerstrasse1";  2 usages
private static final String end = "BaselSpalenring1";  2 usages
```

```
ArrayList<String> path = breadthFirst(start, end);
```

Statt einzelner Knoten werden ganze Pfade in einer FIFO (First in first out) verwaltet. So ist beim Erreichen des Ziels der vollständige Weg direkt verfügbar.

```
private static ArrayList<String> breadthFirst(String start, String end) {  1 usage   Mladen249 +1
    if (start == null || end == null) return null;
    if (!adjList.containsKey(start)) {
        System.out.println("Start node not in adjacency list: " + start);
        return null;
    }

    // FIFO-Struktur mit Pfaden
    ArrayList<ArrayList<String>> paths = new ArrayList<>();
    ArrayList<String> startingPath = new ArrayList<>();
    startingPath.add(start);
    paths.add(startingPath);
```

Ein visited-HashSet verhindert, dass Knoten mehrfach expandiert werden (schützt vor Endlosschleifen, reduziert unnötige Pfaderzeugung und spart Speicher, wodurch die Eigenschaft „kürzester Weg anhand Pfadlänge“ erhalten bleibt).

```
// Globale Besuchsmarkierung (verhindert Endlosschleifen)
Set<String> visited = new HashSet<>();
visited.add(start);
```

Im nächsten Schritt wird in einer Schleife gearbeitet, solange sich Pfade in der Warteschlange befinden. Dabei wird stets der älteste Pfad – also der vorderste Eintrag – entnommen. Er repräsentiert den Weg vom Startpunkt zu einem bestimmten Knoten. Durch das Entfernen am Listenanfang wird die Reihenfolge der Erzeugung beibehalten, was dem FIFO-Prinzip (First In – First Out) der Breitensuche entspricht.

Anschliessend wird der letzte Knoten des Pfads ermittelt, da er den aktuellen Expansionspunkt markiert. Entspricht er dem Ziel, wird der Pfad sofort zurückgegeben und die Suche beendet. Andernfalls werden seine Nachbarn aus der Adjazenzliste geladen. Verfügt der Knoten über keine Nachbarn, etwa bei einer Sackgasse, wird die Schleife mit dem nächsten Pfad fortgesetzt:

```
while (!paths.isEmpty()) {
    ArrayList<String> oldPath = paths.remove( index: 0);
    pathsChecked++;

    String current = oldPath.get(oldPath.size() - 1);

    // Ziel erreicht?
    if (current.equals(end)) return oldPath;

    // Nachbarn holen
    ArrayList<MapData.Destination> connected = adjList.get(current);
    if (connected == null) continue;
```

Jetzt wird jeder Nachbar geprüft. Zuerst wird der Zielknoten aus dem Nachbarobjekt geholt, dann kontrolliert, ob er bereits im Pfad ist oder global als besucht gilt. Dies verhindert Zyklen und doppelte Expansionen. Trifft eins zu, wird der Nachbar übersprungen.

Wenn nicht, wird der bisherige Pfad kopiert und der Nachbar angehängt. So entsteht ein erweiterter Pfad, der am Ende der Warteschlange abgelegt und später verarbeitet wird. Auch wird der Nachbar als besucht markiert und der Zähler der erweiterten Nachbarn erhöht.

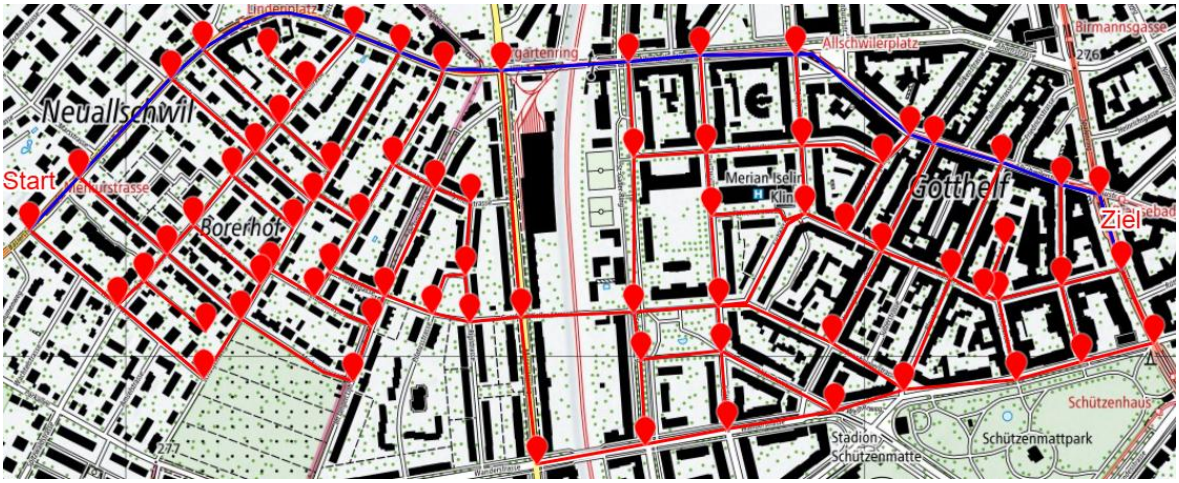
Dann wird geprüft, ob der Nachbar das Ziel ist. Trifft dies zu, wird der erweiterte Pfad sofort zurückgegeben und die Suche beendet, ohne weitere Pfade zu verarbeiten.

```
for (MapData.Destination dest : connected) {
    String next = dest.node();
    // Zyklusbremse: nicht erneut im selben Pfad + nicht erneut global expandieren
    if (oldPath.contains(next)) {
        rejectedCycles++;
        continue;
    }
    if (visited.contains(next)) {
        rejectedCycles++;
        continue;
    }
    ArrayList<String> newPath = (ArrayList<String>) oldPath.clone();
    newPath.add(next);
    paths.add(newPath);
    visited.add(next);
    neighborsVisited++;
    //sofort zurück, falls Ziel gerade gefunden
    if (next.equals(end)) return newPath;
}
```

Resultat

Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
==========================================
Status: Breadth-First Search funktioniert wie erwartet!
------------------------------------------------------------
Benötigte Zeit: 0.716 Millisekunden
Pfadlänge: 17
Distanz: 1751.27 Meter
Anzahl geprüfter Pfade: 69
Anzahl Nachbarn besucht: 72
Abgelehnte (wegen Cycle-Gefahr): 142
------------------------------------------------------------
```



Best-First Search

Der Best-First Search ist ein heuristischer Suchalgorithmus. Als Heuristik wurde die euklidische Distanz zwischen aktuellem und Zielknoten verwendet. Der Algorithmus wählt in jedem Schritt den Knoten, der dem Ziel am nächsten liegt, und ignoriert dabei die bereits zurückgelegte Strecke. Dadurch arbeitet Best-First schnell und ressourcenschonend, kann jedoch suboptimale Ergebnisse liefern, da nur die geschätzte Restdistanz berücksichtigt wird.

```
private static double distanceBetween(String current, String ziel) {  2 usages  Kapischan Sriganthan *

    MapData.GPS lastPos = nodelist.get(current);
    MapData.GPS goalPos = nodelist.get(ziel);

    double xDiff = lastPos.east() - goalPos.east();
    double yDiff = lastPos.north() - goalPos.north();

    return xDiff * xDiff + yDiff * yDiff;
}
```

Funktionsweise

Beim Best-First Search bestimmt die Heuristik – also die geschätzte Restdistanz zum Ziel – die Auswahl der Pfade. Daher wird neben dem Pfad auch die Luftlinie zum Zielknoten des letzten Elements gespeichert. Die Queue enthält somit nicht einzelne Knoten, sondern Path-Objekte mit dem bisherigen Pfad (ArrayList<String>) und der entsprechenden Luftlinie.

Zu Beginn wird eine neue Liste mit dem Startknoten erzeugt, die Luftliniendistanz zum Ziel berechnet und das Objekt in die Queue eingefügt. Dabei wird bewusst nicht die Wurzel gezogen, da nur das Verhältnis und nicht die exakte Distanz relevant ist.

```
public record Path(ArrayList<String> nodes, double distanceToGoal) {  8 usages  Kapischan Sriganthan
}
```

```
private static Path bestFirst(String start, String ziel) {  1 usage  Kapischan Sriganthan
    ArrayList<Path> queue = new ArrayList<>();
    ArrayList<String> startingNode = new ArrayList<>();
    startingNode.add(start);
    Path startPath = new Path(startingNode, distanceBetween(start, ziel));
    queue.add(startPath);
```

```
for (MapData.Destination neighbour : neighbours) {
    if (!oldnodes.contains(neighbour.node())) {
        totalNeighbourVisited++; // zählt alle Nachbarn
        ArrayList<String> newNodes = (ArrayList<String>) oldnodes.clone();
        newNodes.add(neighbour.node());
        queue.add(new Path(newNodes, distanceBetween(neighbour.node(), ziel)));

        if (neighbour.node().equals(ziel)) {
            solutionFound = true;
            break;
        }
    }
```

Die Art, wie Nachbarn zum aktuellen Pfad hinzugefügt werden, entspricht grundsätzlich der Breadth-First Search. Der bestehende Pfad wird geklont und der Nachbar an dessen Ende angehängt. So entsteht für jeden möglichen Schritt ein neuer Pfad, der zurück in die Queue gelegt wird. Im Unterschied zur Breadth-First Search wird jedoch der Pfad mit der geringsten heuristischen Distanz zum Ziel priorisiert und als Nächstes verarbeitet.

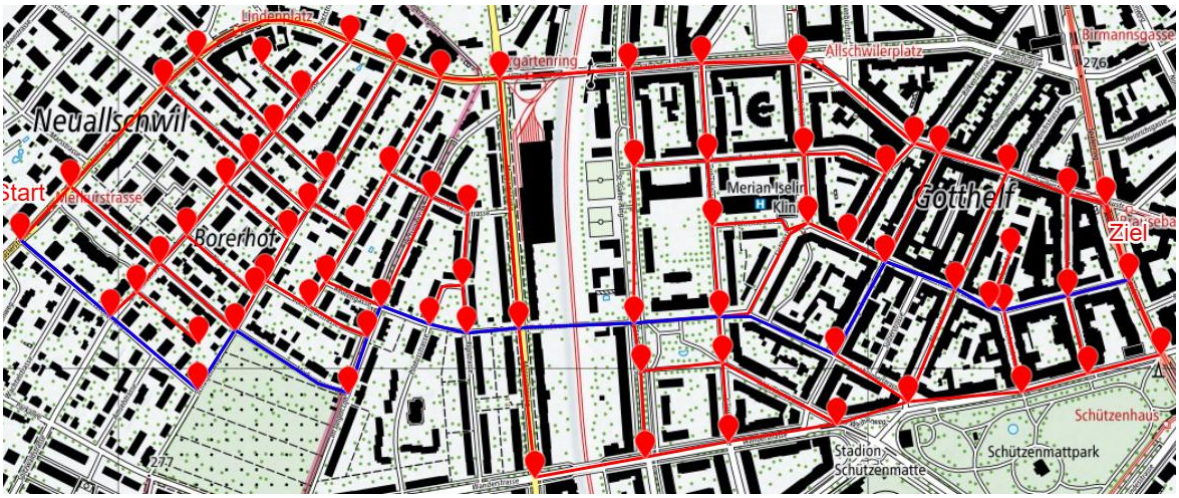
Eine Methode durchsucht dafür alle Pfade in der Queue, vergleicht ihre Distanzwerte und wählt den Index des Pfads mit dem kleinsten Wert:

```
private static int bestPath(ArrayList<Path> queue) {  1 usage   Kapischan Sriganthan
    int bestPath = 0;
    double smallestDistance = queue.get(0).distanceToGoal;

    for (int i = 1; i < queue.size(); i++) {
        if (smallestDistance > queue.get(i).distanceToGoal) {
            bestPath = i;
            smallestDistance = queue.get(i).distanceToGoal;
        }
    }
    return bestPath;
}
```

Resultat

Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
==========================================
Status: BestFirst-Search funktioniert wie erwartet!
------------------------------------------------------------
Benötigte Zeit: 2.005 Millisekunden
Pfadlänge: 19
Distanz:  1'827.18 Meter
Anzahl geprüfter Pfade: 19
Anzahl Nachbarn besucht: 43
Abgelehnte (wegen Cycle-Gefahr): 18
------------------------------------------------------------
```



A* Search

Der A*-Algorithmus ist eine Weiterentwicklung des Best-First Search und zählt zu den effizientesten und zuverlässigsten Suchverfahren.

Im Gegensatz zum Best-First Search berücksichtigt A* neben der geschätzten Restdistanz zum Ziel (Heuristik h(n)) auch die bisher zurückgelegten Kosten g(n).

Dadurch wird bei der Auswahl des nächsten Knotens nicht nur der vielversprechendste, sondern der insgesamt günstigste Pfad priorisiert.

A* arbeitet mit einer Bewertungsfunktion wobei $f(n) = g(n) + h(n)$

g(n) die bisherige Distanz (tatsächliche Kosten)

h(n) die geschätzte Restdistanz (Heuristik, Luftlinie)

darstellen.

In jeder Iteration wählt der Algorithmus den Knoten mit dem kleinsten f-Wert, also der Summe aus realer und geschätzter Distanz. Ist die Heuristik zulässig, garantiert dieses Vorgehen einen optimalen Pfad.

Funktionsweise

Die HashMap searchInfo spielt eine zentrale Rolle. Sie dient als Speicherstruktur für alle bereits besuchten Knoten und hält zu jedem Knoten die aktuell besten bekannten Werte fest. Gespeichert werden:

- bisher zurückgelegte Distanz vom Startknoten (g(n))
- geschätzte Restdistanz zum Zielknoten (h(n))
- Vorgängerknoten, über den der aktuelle Knoten erreicht wurde

Dadurch kann der Algorithmus jederzeit entscheiden, ob ein neu gefundener Pfad zu einem bekannten Knoten kürzer oder länger ist als der bisherige.

```
private static Map<String, NodeInfo> searchInfo = new HashMap<>();  8 usages
```

```
public static class NodeInfo {  7 usages  Kapischan Sriganthan
    double distanceSoFar;  7 usages
    double distanceToGoal;  3 usages
    String previousNode;  3 usages

    NodeInfo(double distanceSoFar, double distanceToGoal, String previousNode) {  2 usages  Kapischan Sriganthan
        this.distanceSoFar = distanceSoFar;
        this.distanceToGoal = distanceToGoal;
        this.previousNode = previousNode;
    }
}
```

Im nächsten Schritt wird geprüft, ob der Nachbarknoten bereits in searchInfo enthalten ist. Falls nicht, wird ein neues NodeInfo-Objekt erstellt und in die Map eingefügt. Existiert der Knoten bereits, wird kontrolliert, ob die neu berechnete Gesamtdistanz – also die Summe aus bisher zurückgelegtem Weg und Distanz zum Nachbarn – kleiner ist als der bisher gespeicherte Wert.

Trifft dies zu, wurde ein kürzerer Pfad gefunden. Der aktuelle Knoten wird als neuer Vorgänger des Nachbarn gespeichert, und die Distanzwerte in searchInfo werden entsprechend aktualisiert:

```
for (MapData.Destination neighbour : neighbours) {
    totalNeighbourVisited++; // zählt alle Nachbarn
    NodeInfo currentNodeInfo = searchInfo.get(currentNode);
    NodeInfo neighbourNodeInfo = searchInfo.get(neighbour.node());
    double distanceNow = currentNodeInfo.distanceSoFar +  neighbour.distance();
    if (searchInfo.keySet().contains(neighbour.node())) {
        if (distanceNow < neighbourNodeInfo.distanceSoFar) {
            neighbourNodeInfo.distanceSoFar = distanceNow;
            neighbourNodeInfo.previousNode = currentNode;
            queue.add(neighbour.node());
        }
        else {
            //Wenn wir den Node schon mal besucht haben und es nicht mehr besser werden kann
            rejectedBecauseOfCircle++;
        }
    } else {
        searchInfo.put(neighbour.node(),
                new NodeInfo(distanceNow, distanceBetween(neighbour.node(), ziel), currentNode));
        queue.add(neighbour.node());
    }
```

Ähnlich wie beim Best-First Search ermittelt auch hier eine Methode den Index des aktuell besten Knotens in der Queue. Im Unterschied dazu basiert die Auswahl jedoch nicht nur auf der Heuristik, sondern zusätzlich auf der bereits zurückgelegten Distanz:

```
private static String findLowestCost() {  2 usages   Kapischan Sriganthan
    double lowestCost = Double.MAX_VALUE;
    String result = null;

    for (String node : queue) {
        NodeInfo nodeInfo = searchInfo.get(node);
        if (lowestCost > nodeInfo.distanceSoFar + nodeInfo.distanceToGoal) {
            lowestCost = nodeInfo.distanceSoFar + nodeInfo.distanceToGoal;
            result = node;
        }
    }
    if (result != null) {
        queue.remove(result);
    }
    return result;
}
```

Die Berechnung der Luftliniendistanz entspricht weitgehend der im Best-First Search. Der einzige Unterschied besteht darin, dass hier die Wurzel aus der Summe der Quadrate der Differenzen gezogen wird. Dies ist notwendig, da beim A*-Algorithmus sowohl die Heuristik als auch die tatsächlich zurückgelegte Strecke in die Gesamtkosten einfliessen.

```
return Math.sqrt(xDiff * xDiff + yDiff * yDiff);
```

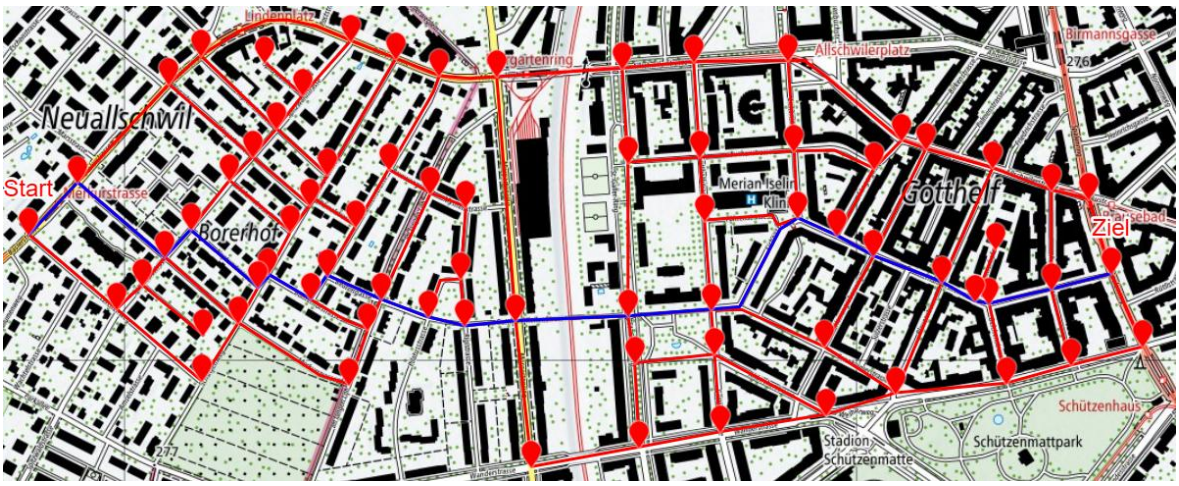
Am Ende der Suche wird die searchInfo-Map genutzt, um den optimalen Pfad zu rekonstruieren. Mithilfe der gespeicherten Vorgängerbeziehungen wird vom Zielknoten rückwärts bis zum Start iteriert. Anschliessend wird die Reihenfolge umgekehrt, sodass der vollständige Pfad in der richtigen Richtung vorliegt. Auf diese Weise kann der kürzeste Pfad effizient und zuverlässig aus den gespeicherten Informationen wiederhergestellt werden.

```java
private static ArrayList<String> constructPath(String start, String ziel) {  1 usage  Kapischan Sriganthan
    ArrayList<String> path = new ArrayList<>();
    path.add(ziel);
    String current = ziel;

    while (!current.equals(start)) {
        NodeInfo nodeInfo = searchInfo.get(current);
        current = nodeInfo.previousNode;
        path.add(current);
    }

    Collections.reverse(path);
    return path;
}
```

Resultat

Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
========================================
Status: A*-Search funktioniert wie erwartet!
----------------------------------------
Benötigte Zeit: 1.967 Millisekunden
Pfadlänge: 22
Distanz:  1'699.77 Meter
Anzahl geprüfter Pfade: 53
Anzahl Nachbarn besucht: 170
Abgelehnte (wegen Cycle-Gefahr): 97
----------------------------------------
```



Phase 3: Neuer oder angepasster Algorithmus

In der dritten Phase wurde das bestehende Projekt um weitere Suchalgorithmen und Optimierungen ergänzt. Ziel war es, die bisher implementierten Ansätze aus Phase 2, um zusätzliche Verfahren zu erweitern und die Realitätsnähe weiter zu prüfen.

Konkret wurden zwei neue Methoden umgesetzt:

- **Dijkstra-Algorithmus** als klassische Variante der optimalen, nicht-heuristischen Suche
- **Beam Search** als heuristische Erweiterung, die mit begrenztem Suchraum arbeitet.

Zusätzlich wurde der A*-Algorithmus um die Heuristik „Speed Limits“ erweitert, die unterschiedliche Geschwindigkeitszonen berücksichtigt und die Wegsuche dadurch realistischer macht.

Dijkstra-Algorithmus

Der Dijkstra-Algorithmus wurde als Referenz für den Vergleich mit A* implementiert. Er arbeitet ohne Heuristik und bestimmt den kürzesten Pfad ausschliesslich anhand der kumulierten Distanzen. Im Gegensatz zu Best-First und A* berücksichtigt er keine geschätzte Restdistanz, sondern nur die tatsächlich zurückgelegte Strecke.

Funktionsweise

Wie bei A* wird auch hier eine HashMap-searchInfo genutzt, um für jeden besuchten Knoten die bislang kürzeste Distanz (distanceSoFar) und den Vorgänger (previousNode) zu speichern.

```
public static class NodeInfo {  6 usages   Kapischan Sriganthan
    double distanceSoFar;  7 usages
    String previousNode;  3 usages

    NodeInfo(double distanceSoFar, String previousNode) {  2 usages   Kapischan Sriganthan
        this.distanceSoFar = distanceSoFar;
        this.previousNode = previousNode;
    }
}
```

Beim Start wird für alle Knoten ein NodeInfo-Objekt angelegt:

- Für den Startknoten mit distanceSoFar = 0,
- für alle anderen mit Double.MAX_VALUE, um unendliche Distanz zu symbolisieren.

```
for (String node : nodelist.keySet()) {
    if (node.equals(start)) {
        searchInfo.put(node, new NodeInfo( distanceSoFar: 0,  previousNode: null));
    } else {
        searchInfo.put(node, new NodeInfo(Double.MAX_VALUE,  previousNode: null));
    }
}
```

Um Zyklen zu vermeiden, werden bereits besuchte Knoten nicht erneut geprüft. Ein HashSet speichert alle verarbeiteten Knoten. Befindet sich ein Nachbar darin, wird er übersprungen, um doppelte Besuche und unnötige Berechnungen zu vermeiden.

```
HashSet<String> visited = new HashSet<>();
```

```
visited.add(currentNode);
```

```
if (!visited.contains(neighbour.node())) {
```

Auch hier wählt eine Methode nach jeder Iteration den Knoten mit der bislang kleinsten Distanz aus der Queue. So wird stets der vielversprechendste Knoten weiterverarbeitet, und der Algorithmus nähert sich schrittweise dem optimalen Pfad.

```
private static String findLowestDistanceSoFar() {  2 usages  Kapischan Sriganthan
    double lowestDistance = Double.MAX_VALUE;
    String result = null;

    for (String node : queue) {
        if (lowestDistance > searchInfo.get(node).distanceSoFar) {
            lowestDistance = searchInfo.get(node).distanceSoFar;
            result = node;
        }
```

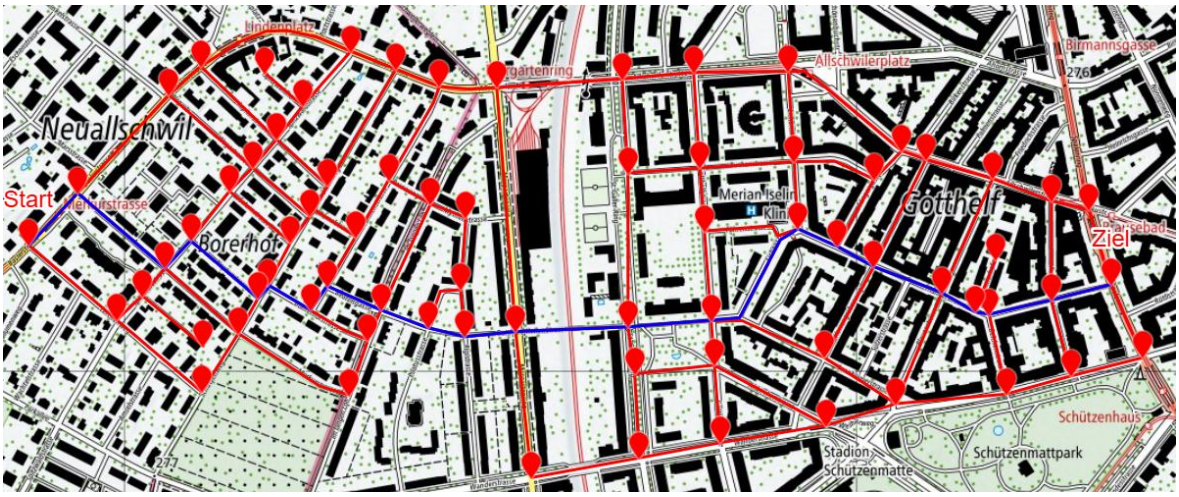
Wird der Knoten mit der geringsten Distanz gefunden und ist noch unbesucht, werden seine Nachbarn geprüft. Ist die neue Gesamtdistanz kleiner, wird der aktuelle Knoten als Vorgänger gespeichert, der Distanzwert aktualisiert und der Nachbar in die Queue eingefügt.

```
for (MapData.Destination neighbour : neighbours) {
    if (!visited.contains(neighbour.node())) {
        totalNeighbourVisited++; // zählt alle Nachbarn
        NodeInfo currentNodeInfo = searchInfo.get(currentNode);
        NodeInfo neighbourNodeInfo = searchInfo.get(neighbour.node());
        double distance = currentNodeInfo.distanceSoFar + neighbour.distance();

        if (neighbourNodeInfo.distanceSoFar > distance) {
            neighbourNodeInfo.distanceSoFar = distance;
            neighbourNodeInfo.previousNode = currentNode;
            queue.add(neighbour.node());
        }
```

Resultat

Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
==========================================
Status: Dijkstra-Algorithmus funktioniert wie erwartet!
----------------------------------------------------------
Benötigte Zeit: 1.066 Millisekunden
Pfadlänge: 22
Distanz:  1'699.77 Meter
Anzahl geprüfter Pfade: 79
Anzahl Nachbarn besucht: 122
Abgelehnte (wegen Cycle-Gefahr): 125
----------------------------------------------------------
```



Beam Search

Die Beam Search wurde ebenfalls in Phase 3 implementiert und stellt eine Kombination aus Best-First Search und Breiten-Suche dar. Im Gegensatz zu Best-First oder A* wird die Suche hier bewusst eingeschränkt, um Rechenzeit und Speicher zu sparen. Dies geschieht über den Parameter Beam Width (BW), der die maximale Anzahl an Pfaden pro Iteration bestimmt, die weiterverfolgt werden dürfen.

Funktionsweise

Beam Search folgt der gleichen Grundstruktur wie Best-First Search: In jeder Iteration werden die Nachbarn der besten Knoten untersucht und nach einer Heuristik, etwa der Luftlinie zum Ziel, bewertet.

Der Unterschied: Nur die besten k Pfade (Beam Width) bleiben erhalten, alle anderen werden verworfen. Das verkleinert den Suchraum und beschleunigt die Suche, kann aber den optimalen Pfad ausschliessen.

Für jeden Knoten wird vorab eine Heuristik berechnet:

```
private static Map<String, Double> heuristics = new HashMap<>();  3 usages
```

```
private static void erstelleHeuristics(String ziel) {  1 usage   Kapischan Sriganthan
    for (String node : nodesList.keySet()) {
        heuristics.put(node, distanceBetween(node, ziel));
    }

}
```

Die Heuristik nutzt die Luftliniendistanz zum Ziel und ähnelt dem Best-First Search. Anders als dort werden jedoch alle Pfade erweitert, bevor nur die besten gemäss Beam Width weiterverfolgt werden.

```
for(ArrayList<String>oldPath : queue) {
totalPathsChecked++; // zählt jeden überprüften Pfad
String current = oldPath.get(oldPath.size()-1);

ArrayList<MapData.Destination> neighbours = adjList.get(current);

for (MapData.Destination neighbour : neighbours) {
    if (!oldPath.contains(neighbour.node())) {
        totalNeighbourVisited++;
        ArrayList<String> newpath = new ArrayList<>(oldPath);
        newpath.add(neighbour.node());
        allcandidates.add(newpath);
    }
    else {
        rejectedBecauseOfCircle++;
    }
    if(neighbour.node().equals(ziel)) {
        return allcandidates.get(allcandidates.size()-1);
    }
```

All diese Pfade landen nach der aktuellen Iteration in einer temporären ArrayList:

```
ArrayList<ArrayList<String>> allcandidates = new ArrayList<>();
```

Anschliessend werden die besten Pfade gemäss der Beam Width ausgewählt und in die zuvor geleerte Queue eingefügt. In der nächsten Iteration werden sie um ihre Nachbarn erweitert und die Kandidatenliste nach der kleinsten Heuristik sortiert.

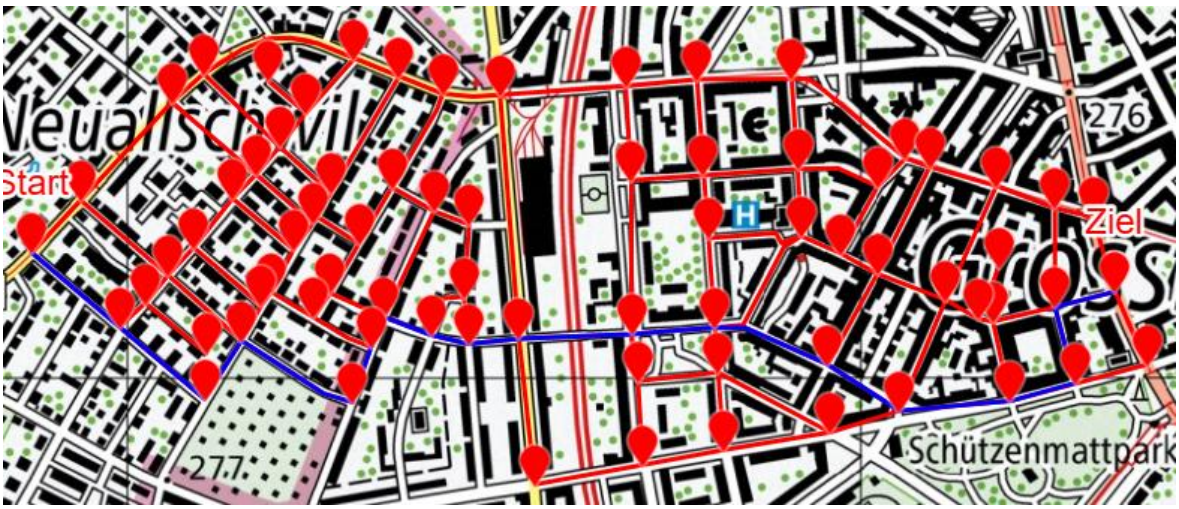
Übersteigt ihre Anzahl die Beam Width, bleiben nur die besten *k* Pfade, sonst wird die gesamte Liste übernommen

```
// Kandidanten nach Heuristik-Wert sortieren
allcandidates.sort(( ArrayList<String> p1,  ArrayList<String> p2) ->
Double.compare(
        heuristics.get(p1.get(p1.size() - 1)),
        heuristics.get(p2.get(p2.size() - 1))
        )
);
```

Resultat

Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
========================================
Status: Beam-Search mit eine Beam-Widht von 3 funktioniert wie erwartet!
------------------------------------------------------------
Benötigte Zeit: 2.353 Millisekunden
Pfadlänge: 18
Distanz:  1'894.38 Meter
Anzahl geprüfter Pfade: 46
Anzahl Nachbarn besucht: 109
Abgelehnte (wegen Cycle-Gefahr): 45
Weggeworfene Paths aufgrund Beam-Width(3): 59
------------------------------------------------------------
```



A* mit neuer Heuristik «Speed Limit»

Der A*-Algorithmus wurde um eine Heuristik erweitert, die auf Fahrzeit statt Distanz optimiert. Dafür erhielt MapData.Destination ein zusätzliches Attribut speedLimit. Auch die Edge-Datei wurde angepasst und um reale Geschwindigkeitsbegrenzungen (20, 30 oder 50 km/h) ergänzt.

```
public record Destination(String node, double distance, int speedLimit) {  36 usages  Kapischan Sriganthan
};
```

Dadurch erhalten wir realistischere Suchergebnisse. Das Ziel ist somit nicht mehr der kürzeste, sondern der schnellste Pfad zwischen Start- und Zielknoten.

Anstelle der bisherigen Distanzen werden nun Zeiten berechnet.

Die Kostenfunktion lautet: $f(n) = g(n) + h(n)$

g(N) = bisher zurückgelegte Zeit (in Sekunden)

h(n) = geschätzte Restzeit zum Ziel (Luftlinie in Sekunden)

Die Zeit wird mit folgender Formel berechnet: `time = distance / (speedLimit / 3.6);`

Da die Distanzen in Metern und die Geschwindigkeit in km/h angegeben sind, wird durch 3.6 geteilt, um in m/s umzurechnen.

Wichtige Codeänderungen:

1. Neue Variablen in der NodeInfo-Klasse

Statt der bisherigen Distanzwerte enthält jedes NodeInfo-Objekt nun die bisherige und geschätzte Zeit:

```
public static class NodeInfo {  7 usages  Kapischan Sriganthan
    double timeSoFar; // Zeit verbraucht bisher  7 usages
    double timeToGoal; // Zeit bis zum Ziel  3 usages
    String previousNode;  3 usages

    NodeInfo(double timeSoFar, double timeToGoal, String previousNode) {  2 usages  Kapischan Sriganthan
        this.timeSoFar = timeSoFar;
        this.timeToGoal = timeToGoal;
        this.previousNode = previousNode;
    }
}
```

2. Berechnung der Zeitkosten im Algorithmus

Beim Durchlaufen der Nachbarn wird für jede Kante die benötigte Zeit ermittelt:

```
// Zeit in Sekunden
double timeNow = currentNodeInfo.timeSoFar +  neighbour.distance() / (neighbour.speedLimit()/3.6);
if (searchInfo.keySet().contains(neighbour.node())) {
    if (timeNow < neighbourNodeInfo.timeSoFar) {
        neighbourNodeInfo.timeSoFar = timeNow;
        neighbourNodeInfo.previousNode = currentNode;
        queue.add(neighbour.node());
    }
    else {
        //Wenn wir den Node schon mal besucht haben und es nicht mehr besser werden kann
        rejectedBecauseOfCircle++;
    }
```

3. Neue Heuristik

Auch die geschätzte Restzeit verwendet die Geschwindigkeit der jeweiligen Kante

```
searchInfo.put(neighbour.node(), // Zeit in Sekunden
        new NodeInfo(timeNow,  timeToGoal: distanceBetween(neighbour.node(), ziel)
                / (neighbour.speedLimit() / 3.6), currentNode));
```

Wirkung der neuen Heuristik:

Durch die Anpassung sucht der Algorithmus nicht mehr den kürzesten, sondern den schnellsten Pfad. So kann eine längere, aber schneller befahrbare Route einer kurzen Nebenstrasse mit niedrigerem Tempolimit vorgezogen werden. Das Ergebnis entspricht damit einer realistischeren Routenplanung.

Resultat

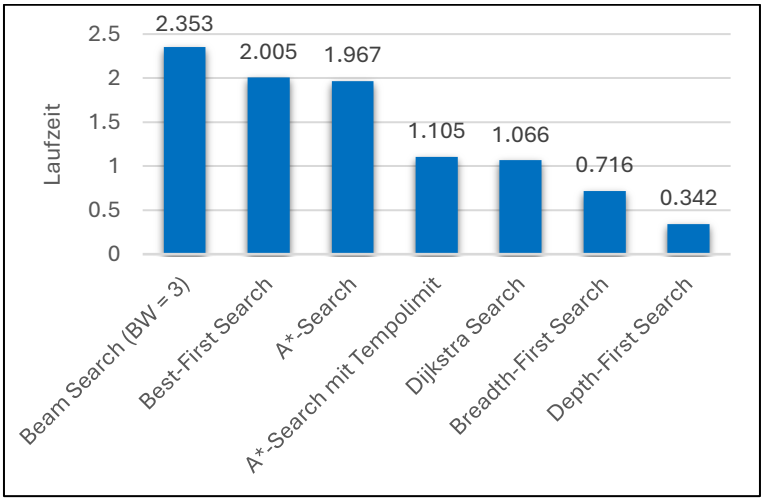
Ausgehend vom Punkt «AllschwilBaslerstrasse1» mit dem Ziel «BaselSpalenring1» wurden folgende Ergebnisse geliefert:

```
========================================
Status: A*-Search funktioniert wie erwartet!
----------------------------------------
Benötigte Zeit: 1.105 Millisekunden
Pfadlänge: 17
Gesamtzeit: 2.10 Minuten
Distanz:  1'751.27 Meter
Anzahl geprüfter Pfade: 16
Anzahl Nachbarn besucht: 46
Abgelehnte (wegen Cycle-Gefahr): 15
----------------------------------------
```

Vergleich und Bewertung der Algorithmen

Folgend werden die implementierten Algorithmen gegenübergestellt und anhand ihrer Ergebnisqualität systematisch bewertet

Laufzeit in Millisekunden:

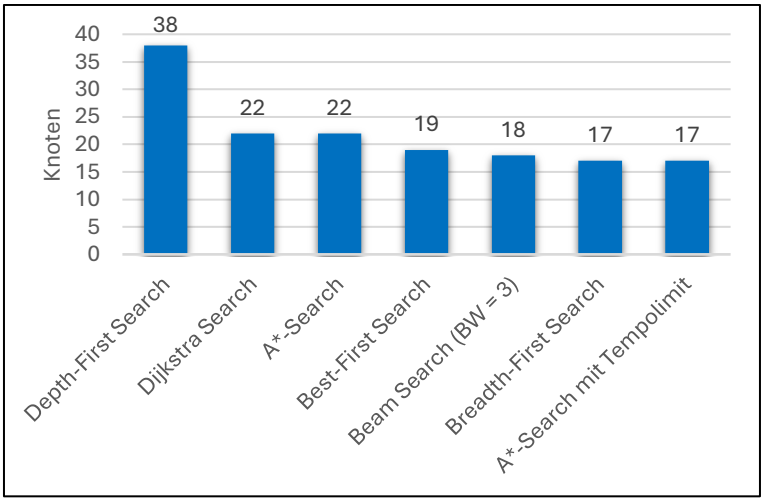


Am schnellsten: DFS und BFS

Am langsamsten: Best-First und Beam (BW=3)

Die einfachen Verfahren schliessen aufgrund ihrer Simplizität schneller ab.

Pfadlänge in Knoten:



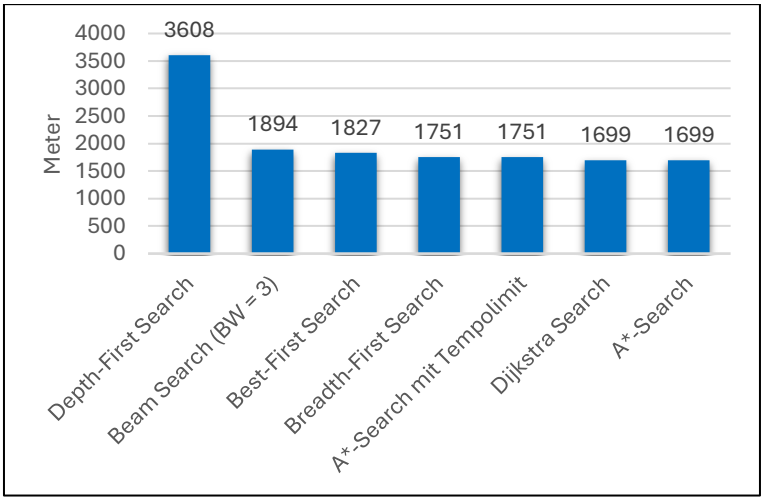
Am kürzesten: A* (km/h) und BFS

Am längsten: DFS und Dijkstra

Kürzere Ketten deuten auf zielgerichtete Suche hin

DFS durchläuft viele Umwege aufgrund der Simplizität.

Gesamtlänge in Meter:

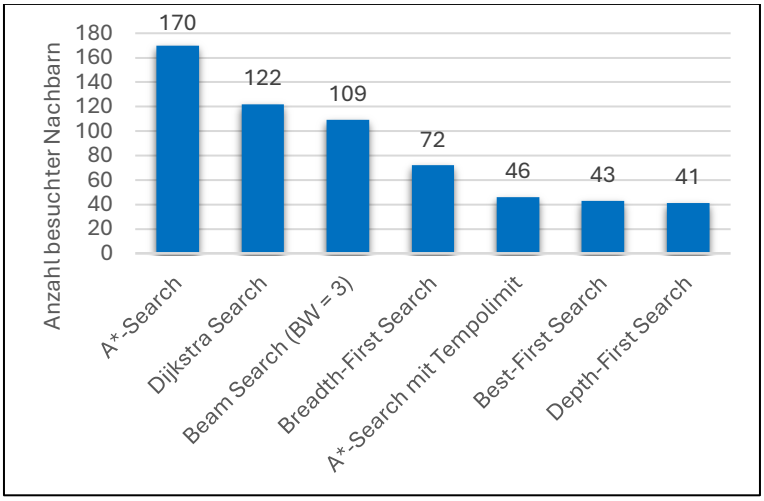


Am-kürzesten: A* und Dijkstra

Am-längsten: DFS und Beam (BW=3)

Distanzbasierte Verfahren liefern die kompaktesten Routen; A* (km/h) optimiert auf Fahrzeit, nicht zwingend auf Strecke.

Anzahl besuchter Nachbarn:

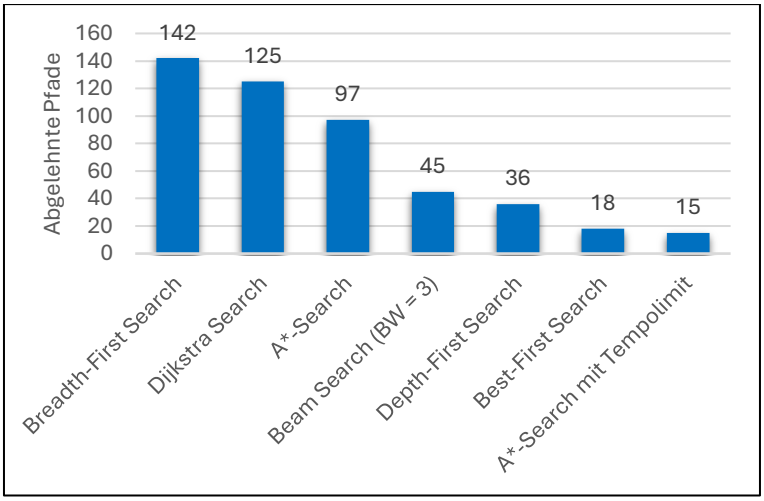


Am-wenigsten: **DFS** und **Best-First**

Am-meisten: **A*** und **Dijkstra**

Viele Expansionen bedeuten breitere Suche/Overhead; die Zeit-Heuristik A*(km/h) fokussiert die Suche zusätzlich. Einfache Verfahren haben weniger aufgrund ihrer Simplizität.

Abgelehnte Pfade:



Am-wenigsten: **A* (km/h)** und **Best-First**

Am-meisten: **BFS** und **Dijkstra**

Breite, uninformierte Suche verwirft mehr Kandidaten; Heuristiken tendieren unnötige Abzweige zu reduzieren.