A* POI-Aware Pathfinding Plugin - Technische Dokumentation

Inhaltsverzeichnis

Übersicht und Funktionsweise

Dieses Neo4j-Plugin implementiert einen erweiterten A*-Algorithmus, der speziell für barrierefreie Routenplanung entwickelt wurde. Der klassische A*-Algorithmus findet den kürzesten Weg zwischen zwei Punkten, indem er eine Kostenfunktion minimiert. Diese Implementierung erweitert das Konzept erheblich durch die Integration mehrerer Faktoren, die für Menschen mit eingeschränkter Mobilität relevant sind.

Der Kern-Algorithmus berechnet für jedes Straßensegment eine Kostenfunktion, die aus mehreren Komponenten besteht. Die Grundformel lautet: Segment-Kosten gleich Länge multipliziert mit Steigungsgewicht multipliziert mit Oberflächengewicht minus POI-Bonus. Diese Formel wird für jedes einzelne Segment im Graphen angewendet, während der Algorithmus den optimalen Pfad sucht.

Im Gegensatz zu einfachen Routing-Algorithmen berücksichtigt diese Implementierung, dass nicht jeder Meter gleich schwer zu bewältigen ist. Eine steile Steigung kann für Rollstuhlfahrer oder Menschen mit Gehbehinderung mehrfach schwerer sein als die gleiche Distanz auf flachem Terrain. Unterschiedliche Oberflächen wie Kopfsteinpflaster oder unbefestigte Wege bedeuten zusätzliche Anstrengung. Gleichzeitig können Points of Interest wie Toiletten oder Sitzbänke eine Route attraktiver machen, selbst wenn sie etwas länger ist.

Der A*-Algorithmus nutzt eine Heuristik, um die Suche zu beschleunigen. In dieser Implementierung wird die euklidische Distanz (Luftlinie) zum Ziel als Heuristik verwendet. Diese Heuristik ist "admissible", das heißt sie überschätzt niemals die tatsächlichen Kosten, was garantiert, dass der optimale Pfad gefunden wird.

Prozedur-Aufruf und Rückgabewerte

Der Prozedur-Aufruf erfolgt in Cypher mit der folgenden Syntax:

```cypher
CALL org.kiss.flexibleWeightedShortestPath(
    startNodeId,
    endNodeId,
    xProp,
    yProp,
    slopeCategories,
    propertyWeights,
    excludes,
    poiWeights
)
YIELD path, totalDistance, weightedDistance, poiCount, poiDetails, warnings, routeQuality
```

Die Prozedur gibt mehrere Werte zurück, die unterschiedliche Aspekte der berechneten Route beschreiben.

path ist eine Liste aller Knoten, die die berechnete Route bilden. Dies sind die tatsächlichen Neo4j-Node-Objekte in der Reihenfolge von Start bis Ziel. Diese Liste kann direkt verwendet werden, um die Route zu visualisieren oder weitere Analysen durchzuführen.

totalDistance gibt die physische Gesamtdistanz der Route in Metern an. Dies ist die ungewichtete, reale Länge der Strecke, wie sie auf einer Karte gemessen würde. Dieser Wert summiert einfach alle "length"-Properties der durchlaufenen Relationships.

weightedDistance ist die gewichtete Distanz, die alle Faktoren (Steigung, Oberfläche) berücksichtigt. Diese Zahl ist typischerweise höher als totalDistance und repräsentiert die "gefühlte" oder "effektive" Distanz unter Berücksichtigung aller Erschwernisse. Beispielsweise könnten 1000 Meter physische Distanz durch steile Steigungen und schlechte Oberflächen zu 2500 Metern gewichteter Distanz werden.

poiCount ist einfach die Anzahl der gefundenen POIs entlang der Route. Dies gibt einen schnellen Überblick darüber, wie viele interessante Punkte entlang des Weges liegen.

poiDetails ist eine Liste von Maps, wobei jede Map detaillierte Informationen über einen gefundenen POI enthält. Jede Map enthält typischerweise die Felder "type" (z.B. "toilets"), "osm_id" (OpenStreetMap-ID), "walk_distance" (Gehentfernung vom Weg zum POI), "intersection_id" (welche Kreuzung in der Nähe ist) und optional "location" (geografische Koordinaten).

warnings ist eine Liste von String-Nachrichten, die dem Benutzer mitteilen, wenn bestimmte Anforderungen nicht erfüllt werden konnten. Diese Warnungen helfen zu verstehen, warum eine Route möglicherweise nicht ideal ist. Beispiele wären "POI spacing not optimal but sufficient amenities found" oder "Only 1 toilet(s) found (needed: 2)".

routeQuality ist ein String-Indikator, der die Gesamtqualität der Route beschreibt. Es gibt drei mögliche Werte: "perfect" bedeutet, dass alle Anforderungen vollständig erfüllt sind. "good" bedeutet, dass kleinere Abstriche gemacht wurden, die Route aber brauchbar ist. "fallback" bedeutet, dass wichtige Anforderungen nicht erfüllt werden konnten und die beste verfügbare Alternative zurückgegeben wird.

Parameter 1: startNodeId

Typ: String (Neo4j Element-ID)

Pflichtparameter: Ja

Beschreibung:

Die startNodeId ist die eindeutige Element-ID des Knotens, von dem die Route starten soll. In Neo4j Version 5 und höher werden Element-IDs verwendet, die ein spezifisches Format haben. Eine typische Element-ID sieht aus wie "4:a1b2c3d4-e5f6-7890-abcd-ef1234567890:0". Diese IDs sind datenbank-intern und unterscheiden sich von den älteren Legacy-Node-IDs, die einfache Zahlen waren.

Die Element-ID wird intern mit der Methode tx.getNodeByElementId() aufgelöst, um das tatsächliche Node-Objekt zu erhalten. Wenn die angegebene Element-ID nicht existiert oder nicht zu einem Knoten im Graphen führt, wird eine NotFoundException geworfen und die Prozedur schlägt fehl.

Wichtige Hinweise:

Der Startknoten muss Teil des Routing-Netzwerks sein, das heißt er muss über Relationships mit anderen Knoten verbunden sein. Ein isolierter Knoten ohne Verbindungen kann nicht als Startpunkt verwendet werden, da keine Route von ihm ausgehen kann.

Der Startknoten sollte die Properties haben, die in xProp und yProp spezifiziert sind, damit die Heuristik-Berechnung funktioniert. Wenn diese Properties fehlen, werden Defaultwerte (0.0) verwendet, was zu suboptimalen Pfaden führen kann.

Beispiel:

Angenommen, Sie haben einen Knoten mit der Element-ID "4:abc123:0", dann würde der Aufruf beginnen mit:

```cypher
```

```
CALL org.kiss.flexibleWeightedShortestPath(
  "4:abc123:0",  // startNodeId
  ...
)
```

Um die Element-ID eines Knotens zu finden, können Sie eine einfache Cypher-Abfrage verwenden:

```cypher
MATCH (n:Intersection {name: "Hauptbahnhof"})
RETURN elementId(n)
```

Parameter 2: endNodeId

Typ: String (Neo4j Element-ID)

Pflichtparameter: Ja

Beschreibung:

Die endNodeId ist die eindeutige Element-ID des Zielknotens, zu dem die Route führen soll. Alle Eigenschaften und Anforderungen, die für startNodeId gelten, gelten auch für endNodeId.

Wichtige Hinweise:

Start- und Endknoten müssen über das Routing-Netzwerk miteinander verbunden sein. Wenn keine Verbindung existiert oder alle möglichen Wege durch die excludes-Parameter blockiert sind, gibt die Prozedur ein leeres Ergebnis zurück (Stream.empty()). Dies ist keine Fehlermeldung, sondern bedeutet einfach, dass keine gültige Route gefunden wurde.

Die Distanz zwischen Start und Ziel beeinflusst die Performance erheblich. Kurze Routen (unter 1 km) werden typischerweise in unter 50 Millisekunden berechnet, während lange Routen (über 5 km) mehrere Sekunden benötigen können, insbesondere wenn POI-Suche aktiviert ist.

Zusammenspiel mit Heuristik:

Der Endknoten ist entscheidend für die Heuristik-Berechnung. Der A*-Algorithmus schätzt für jeden untersuchten Knoten, wie weit dieser noch vom Ziel entfernt ist. Diese Schätzung basiert auf den Koordinaten (xProp, yProp) von beiden Knoten. Je genauer diese Koordinaten sind, desto effizienter arbeitet der Algorithmus.

Beispiel:

```cypher
```

```
CALL org.kiss.flexibleWeightedShortestPath(
  "4:abc123:0",  // startNodeId
  "4:def456:0",  // endNodeId
  ...
)
```

Parameter 3: xProp

Typ: String (Property-Name)

Pflichtparameter: Ja

Beschreibung:

Der xProp-Parameter definiert den Namen der Property, die die X-Koordinate (typischerweise Longitude bzw. geografische Länge) eines Knotens speichert. Diese Property muss bei allen Knoten im Routing-Netzwerk vorhanden sein, da sie für die Heuristik-Berechnung verwendet wird.

Verwendung im Algorithmus:

Der Hauptzweck von xProp ist die Berechnung der Heuristik, also der geschätzten Restdistanz zum Ziel. Die Heuristik-Funktion berechnet die euklidische Distanz (Luftlinie) zwischen zwei Knoten mit der Formel: Wurzel aus ((x₂ minus x₁) zum Quadrat plus (y₂ minus y₁) zum Quadrat). Diese Distanz dient als untere Schranke für die tatsächlichen Kosten zum Ziel.

Koordinatensysteme:

Die Wahl des Koordinatensystems ist wichtig für die Genauigkeit. Mögliche Optionen sind:

WGS84 Dezimalgrad (latitude/longitude): Dies ist das Standard-GPS-Format. X-Werte sind typischerweise zwischen -180 und 180, Y-Werte zwischen -90 und 90. Vorteil ist die direkte Kompatibilität mit GPS-Daten. Nachteil ist, dass die euklidische Distanz nur eine Approximation der tatsächlichen Erdentfernung ist, was aber für die Heuristik ausreichend ist.

UTM-Koordinaten: Universal Transverse Mercator Koordinaten sind metrisch und erlauben präzise Distanzberechnungen. X- und Y-Werte sind typischerweise sechsstellige Zahlen in Metern. Dies ist ideal für lokale Anwendungen in einem begrenzten geografischen Bereich.

Lokale kartesische Koordinaten: Für Innenraum-Navigation oder kleine Bereiche können auch lokale metrische Koordinatensysteme verwendet werden.

Konsistenz ist kritisch:

Alle Knoten im Graphen müssen das gleiche Koordinatensystem verwenden. Ein Mix aus verschiedenen Systemen führt zu völlig falschen Heuristik-Werten und damit zu suboptimalen oder fehlerhaften Routen.

Fehlende Properties:

Wenn ein Knoten die xProp-Property nicht hat, wird der Defaultwert 0.0 verwendet. Dies führt zu einer fehlerhaften Heuristik für diesen Knoten, was die Performance reduziert und möglicherweise zu suboptimalen Pfaden führt, da der Algorithmus die Distanz nicht korrekt schätzen kann.

Beispiel:

Wenn Ihre Knoten Longitude in einer Property namens "lon" speichern:

```cypher
CALL org.kiss.flexibleWeightedShortestPath(
    startNodeId,
    endNodeId,
    "lon",  // xProp
    "lat",  // yProp entsprechend
    ...
)
```

Oder bei UTM-Koordinaten:

```cypher
CALL org.kiss.flexibleWeightedShortestPath(
    startNodeId,
    endNodeId,
    "utm_east",  // xProp
    "utm_north", // yProp entsprechend
    ...
)
```

Parameter 4: yProp

Typ: String (Property-Name)

Pflichtparameter: Ja

Beschreibung:

Der yProp-Parameter definiert den Namen der Property, die die Y-Koordinate (typischerweise Latitude bzw. geografische Breite) eines Knotens speichert. Alle Erklärungen, Anforderungen und Hinweise, die für xProp gelten, gelten analog auch für yProp.

Zusammenspiel mit xProp:

xProp und yProp bilden zusammen das Koordinatenpaar, das jeden Knoten im zweidimensionalen Raum positioniert. Die Heuristik-Funktion verwendet beide Koordinaten gleichberechtigt in der euklidischen Distanzformel. Es ist daher essentiell, dass beide Properties im gleichen Koordinatensystem vorliegen und bei allen Knoten vorhanden sind.

Typische Property-Namen:

In der Praxis werden häufig folgende Namenskonventionen verwendet: "y" und "x" für generische Koordinaten, "lat" und "lon" oder "latitude" und "longitude" für WGS84-Koordinaten, "utm_north" und "utm_east" für UTM-Koordinaten, "northing" und "easting" als alternative UTM-Bezeichnung.

Die konkrete Namensgebung ist flexibel und hängt von Ihrer Datenstruktur ab. Wichtig ist nur, dass die Namen konsistent im gesamten Graphen verwendet werden und korrekt als Parameter übergeben werden.

Parameter 5: slopeCategories

Typ: Liste von Maps (Array von Steigungskategorie-Objekten)

Pflichtparameter: Ja

Beschreibung:

Der slopeCategories-Parameter ist einer der komplexesten und wichtigsten Parameter des Algorithmus. Er definiert, wie verschiedene Steigungen die Routenkosten beeinflussen. Steigungen sind für Menschen mit eingeschränkter Mobilität oft das größte Hindernis, weshalb ihre korrekte Gewichtung entscheidend für die Routenqualität ist.

Konzept der Steigungskategorien:

Eine Steigungskategorie beschreibt einen Bereich von Steigungen und definiert, wie stark Wege in diesem Bereich gewichtet werden sollen. Jede Kategorie hat vier Eigenschaften:

"name" ist ein beschreibender Name für die Kategorie, etwa "flach", "leichte_steigung", "mittel" oder "steil". Dieser Name dient nur der Dokumentation und wird intern nicht verwendet. Er hilft aber beim Verständnis und bei der Wartung der Konfiguration.

"maxSlope" definiert die maximale Steigung für diese Kategorie in Prozent. Beispielsweise bedeutet maxSlope 3.0, dass diese Kategorie für alle Steigungen bis einschließlich 3 Prozent gilt. Die Steigung wird als Prozent berechnet: (Höhenunterschied dividiert durch horizontale Distanz) multipliziert mit 100. Eine Steigung von 5 Prozent bedeutet, dass auf 100 Meter horizontaler Strecke 5 Meter Höhenunterschied überwunden werden.

"upWeight" ist der Gewichtungsfaktor für bergauf führende Segmente. Ein Wert von 1.0 bedeutet neutrale Gewichtung (Segment zählt normal). Werte größer als 1.0 bedeuten, dass das Segment schwerer zu bewältigen ist und daher stärker in die Kosten eingeht. Beispielsweise bedeutet upWeight 2.0, dass jeder Meter bergauf doppelt so schwer zählt wie ein Meter auf flachem Terrain.

"downWeight" ist der Gewichtungsfaktor für bergab führende Segmente. Typischerweise ist bergab leichter als bergauf, weshalb downWeight oft kleiner als upWeight ist. Allerdings kann für Rollstuhlfahrer steiles bergab ebenfalls problematisch sein wegen der Bremsproblematik, was in der Gewichtung berücksichtigt werden sollte.

Matching-Logik im Detail:

Der Algorithmus verarbeitet die Kategorien in der Reihenfolge, in der sie übergeben werden. Für jedes Straßensegment wird die Steigung aus der "slope"-Property gelesen. Dann wird die Liste der Kategorien durchlaufen, und die erste Kategorie, deren maxSlope größer oder gleich dem Absolutwert der Segment-Steigung ist, wird verwendet.

Beispiel: Gegeben sind Kategorien mit maxSlope 3.0, 8.0 und 15.0 in dieser Reihenfolge. Ein Segment hat eine Steigung von 5.2 Prozent. Der Algorithmus prüft: Ist 5.2 kleiner gleich 3.0? Nein. Ist 5.2 kleiner gleich 8.0? Ja. Also wird die zweite Kategorie (maxSlope 8.0) verwendet. Das bedeutet: Die Reihenfolge der Kategorien bestimmt die Priorität. Kategorien sollten aufsteigend nach maxSlope sortiert sein.

Bergauf versus Bergab Unterscheidung:

Nach der Auswahl der Kategorie wird geprüft, ob die Steigung positiv (bergauf) oder negativ (bergab) ist. Bei positiver Steigung wird upWeight verwendet, bei negativer Steigung downWeight. Das Vorzeichen der Steigung ist also entscheidend: Eine "slope" von +5.0 bedeutet 5 Prozent bergauf, eine "slope" von -5.0 bedeutet 5 Prozent bergab. Beide würden zur gleichen Kategorie führen (da Absolutwert), aber unterschiedliche Gewichte erhalten.

Praktische Gewichtungs-Strategie:

Für barrierefreies Routing sind typische Strategien:

Für Rollstuhlnutzung: Flache Bereiche (0 bis 3 Prozent) nahezu neutral gewichten (upWeight 1.0 bis 1.1). Mittlere Steigungen (3 bis 6 Prozent) deutlich höher gewichten (upWeight 2.0 bis 3.0), da sie schwer zu überwinden sind. Starke Steigungen (über 6 Prozent) extrem hoch gewichten (upWeight 5.0 bis 10.0) oder sogar vermeiden, da sie für Rollstühle oft unpassierbar sind. Bergab kann bei Rollstühlen ebenfalls problematisch sein (Bremsen erforderlich), daher downWeight oft höher als 1.0, etwa 1.2 bis 1.5 auch bei moderaten Gefällen.

Für Fußgänger mit Gehbehinderung: Flache Bereiche neutral (upWeight 1.0), leichte Steigungen moderat erhöhen (upWeight 1.3 bis 1.5), starke Steigungen deutlich erhöhen (upWeight 2.0 bis 3.0). Bergab ist meist weniger problematisch, daher downWeight oft unter 1.0, etwa 0.9 bis 1.0.

Für normale Fußgänger: Alle Steigungen relativ gleichmäßig gewichten, etwa upWeight 1.0 bis 1.1 und downWeight 0.95 bis 1.0, da Steigungen zwar anstrengender sind, aber kein großes Hindernis darstellen.

Fallback-Kategorie:

Es ist essentiell, eine letzte Kategorie mit einem sehr hohen maxSlope (z.B. 999.0) zu definieren, die als Fallback für alle extremen Steigungen dient. Ohne diese Fallback-Kategorie würden Segmente mit sehr steilen Steigungen den Defaultwert 1.0 erhalten, was inkorrekt wäre.

Beispiel einer vollständigen Konfiguration:

```json
[
  {
    "name": "flach",
    "maxSlope": 3.0,
    "upWeight": 1.0,
    "downWeight": 0.95
  },
  {
    "name": "leichte_steigung",
    "maxSlope": 6.0,
    "upWeight": 1.4,
    "downWeight": 1.0
  },
  {
    "name": "mittlere_steigung",
    "maxSlope": 10.0,
    "upWeight": 2.5,
    "downWeight": 1.2
  },
  {
    "name": "steile_steigung",
    "maxSlope": 999.0,
    "upWeight": 10.0,
    "downWeight": 2.0
  }
]
```

Diese Konfiguration würde bedeuten: Wege bis 3 Prozent Steigung zählen fast normal. Wege mit 3 bis 6 Prozent sind bergauf 40 Prozent schwerer. Wege mit 6 bis 10 Prozent sind bergauf 2.5-mal schwerer. Alles über 10 Prozent wird extrem stark gewichtet und damit praktisch vermieden.

Parameter 6: propertyWeights

Typ: Map von Maps (Verschachtelte Map-Struktur)

Pflichtparameter: Ja

Beschreibung:

Der propertyWeights-Parameter definiert, wie verschiedene Eigenschaften von Straßensegmenten (Relationships) die Routenkosten beeinflussen. Dies ermöglicht die Berücksichtigung von Faktoren wie

Oberflächenbeschaffenheit, Breite, Beleuchtung oder anderen relevanten Eigenschaften.

Struktur im Detail:

Die äußere Map hat als Schlüssel die Namen von Relationship-Properties, zum Beispiel "surface", "smoothness", "width" oder "lit". Der Wert jedes Schlüssels ist wiederum eine Map, die mögliche Werte dieser Property auf Gewichtungsfaktoren abbildet.

Beispiel für die "surface"-Property:

```json
"surface": {
  "asphalt": 1.0,
  "paved": 1.05,
  "concrete": 1.0,
  "paving_stones": 1.3,
  "cobblestone": 2.5,
  "gravel": 3.5,
  "unpaved": 6.0
}
```

Dies bedeutet: Wenn ein Straßensegment die Property "surface" mit dem Wert "asphalt" hat, wird ein Gewicht von 1.0 angewendet (neutral, Referenzwert). Hat das Segment "surface" mit Wert "cobblestone", wird ein Gewicht von 2.5 angewendet (Segment ist 2.5-mal schwerer zu bewältigen).

Multiplikative Kombination:

Ein wichtiges Konzept ist, dass alle gefundenen Gewichte miteinander multipliziert werden. Wenn ein Segment mehrere Properties hat, die alle in propertyWeights definiert sind, werden deren Gewichte kombiniert.

Beispiel: Ein Segment hat "surface": "paving_stones" (Gewicht 1.3) und "smoothness": "bad" (Gewicht 2.0). Das Gesamtgewicht für Properties ist 1.3 mal 2.0 gleich 2.6. Das Segment wird also 2.6-mal schwerer gewichtet als ein Segment mit optimalen Properties.

Diese multiplikative Kombination kann zu sehr hohen Gewichten führen, wenn mehrere ungünstige Faktoren zusammenkommen. Ein Segment mit schlechter Oberfläche, schlechter Smoothness und schmaler Breite könnte leicht ein Gesamtgewicht von 5.0 oder höher erreichen.

Fehlende Properties:

Wenn ein Segment eine Property nicht hat, die in propertyWeights definiert ist, wird für diese Property der Gewichtungsfaktor 1.0 angenommen (neutral). Das bedeutet: Nur explizit vorhandene Properties beeinflussen das Gewicht. Fehlende Informationen werden als "durchschnittlich" oder "unbekannt" behandelt.

Nicht definierte Werte:

Wenn ein Segment einen Property-Wert hat, der nicht in der entsprechenden Gewichts-Map vorkommt, wird ebenfalls 1.0 (neutral) verwendet. Beispiel: Wenn "surface": "tiles" im Graphen vorkommt, aber nicht in der propertyWeights-Definition, wird dieses Segment neutral gewichtet für die surface-Property.

Typische Properties und deren Bedeutung:

surface (Oberflächenbeschaffenheit): Dies ist meist die wichtigste Property. Unterschiedliche Oberflächen haben dramatisch unterschiedliche Auswirkungen auf die Befahrbarkeit. Asphalt und Beton sind ideal (Gewicht 1.0). Gepflasterte Wege (paved, paving_stones) sind noch gut, aber etwas holpriger (Gewicht 1.1 bis 1.3). Kopfsteinpflaster (cobblestone) ist für Rollstühle sehr schwierig (Gewicht 2.0 bis 3.0). Kies (gravel) ist noch schwieriger (Gewicht 3.0 bis 4.0). Unbefestigte Wege (unpaved, dirt, grass) sind oft unpassierbar (Gewicht 5.0 bis 10.0 oder via excludes ausschließen).

smoothness (Ebenheit): Beschreibt, wie glatt die Oberfläche ist. Werte sind typischerweise "excellent", "good", "intermediate", "bad", "very_bad". Excellent und good sind ideal (Gewicht 0.9 bis 1.0). Intermediate erfordert mehr Aufmerksamkeit (Gewicht 1.2 bis 1.5). Bad ist deutlich schwieriger (Gewicht 2.0 bis 3.0). Very_bad ist oft unpassierbar (Gewicht 5.0 oder höher).

width (Breite): Schmale Wege können für Rollstühle problematisch sein. Typische Kategorien: Breite über 2 Meter ist ideal (Gewicht 1.0). Breite 1.5 bis 2 Meter ist noch gut (Gewicht 1.1). Breite 1 bis 1.5 Meter ist eng (Gewicht 1.5). Breite unter 1 Meter ist sehr schwierig (Gewicht 3.0 oder ausschließen).

lit (Beleuchtung): Relevant für Routing bei Dunkelheit. "yes" für beleuchtete Wege (Gewicht 1.0), "no" für unbeleuchtete Wege (Gewicht 1.2 bis 1.5 bei Nacht-Routing).

incline (zusätzlich zu slope): Manche Datenmodelle haben separate incline-Angaben. Diese können ähnlich wie slope gewichtet werden.

Strategie für Gewichtswahl:

Die Gewichte sollten die relative Schwierigkeit oder den Komfortunterschied widerspiegeln. Ein guter Ansatz ist: Definieren Sie einen Referenzwert (typischerweise asphalt oder paved mit Gewicht 1.0). Überlegen Sie für jeden anderen Wert: Wie viel schwerer oder leichter ist dieser im Vergleich zur Referenz? Seien Sie bei kritischen Unterschieden eher konservativ mit extremen Gewichten, da diese schnell zu sehr eingeschränkten Routenoptionen führen.

Beispiel einer umfassenden Konfiguration:

```
json
```

```
{
  "surface": {
    "asphalt": 1.0,
    "concrete": 1.0,
    "paved": 1.05,
    "paving_stones": 1.2,
    "sett": 2.0,
    "cobblestone": 2.5,
    "compacted": 1.8,
    "gravel": 3.5,
    "unpaved": 6.0,
    "dirt": 7.0,
    "grass": 8.0
  },
  "smoothness": {
    "excellent": 0.9,
    "good": 1.0,
    "intermediate": 1.3,
    "bad": 2.5,
    "very_bad": 5.0,
    "horrible": 10.0
  },
  "width_category": {
    "wide": 1.0,
    "normal": 1.1,
    "narrow": 1.5,
    "very_narrow": 2.5
  }
}
```

Parameter 7: excludes

Typ: Map von Property-Namen zu Listen von auszuschließenden Werten

Pflichtparameter: Ja (kann aber leer sein: {})

Beschreibung:

Der excludes-Parameter definiert harte Ausschlusskriterien für Relationships. Während propertyWeights bestimmt, wie schwer ein Weg ist, bestimmt excludes, welche Wege überhaupt nicht verwendet werden dürfen. Dies ist wichtig für absolute No-Go-Bereiche wie Treppen für Rollstuhlfahrer oder Privatwege.

Funktionsweise im Detail:

Für jedes Straßensegment (Relationship) prüft der Algorithmus, ob es eine der ausgeschlossenen Eigenschaften hat. Die Prüfung erfolgt für jeden Eintrag in der excludes-Map. Wenn die Relationship die angegebene Property hat UND der Wert dieser Property in der Liste der auszuschließenden Werte vorkommt, wird die Relationship komplett übersprungen und nicht für das Routing verwendet.

Dies geschieht bereits in der Frühphase der Pfadsuche, das heißt ausgeschlossene Relationships werden niemals in die Kostenkalkulation einbezogen. Sie sind für den Algorithmus praktisch nicht existent.

Struktur:

```
json

{
  "highway": ["steps", "track"],
  "access": ["private", "no"],
  "wheelchair": ["no", "limited"]
}
```

Diese Konfiguration bedeutet: Jede Relationship, die die Property "highway" mit Wert "steps" oder "track" hat, wird ausgeschlossen. Zusätzlich wird jede Relationship mit "access" gleich "private" oder "no" ausgeschlossen. Zusätzlich wird jede Relationship mit "wheelchair" gleich "no" oder "limited" ausgeschlossen.

Logik-Verknüpfung:

Die einzelnen Kriterien sind ODER-verknüpft innerhalb einer Property-Liste und ebenfalls ODER-verknüpft zwischen verschiedenen Properties. Das heißt: Ein Segment wird ausgeschlossen, wenn es IRGENDEINS der Kriterien erfüllt. Es reicht, wenn eine einzige Bedingung zutrifft.

Beispiel: Eine Relationship mit "highway": "steps" wird ausgeschlossen, auch wenn "access" auf "yes" steht. Die Treppe ist ein Ausschlusskriterium, unabhängig von anderen Properties.

Wichtige Anwendungsfälle:

Für Rollstuhl-Routing ist der wichtigste Ausschluss "highway": ["steps"], da Treppen absolut unpassierbar sind. Zusätzlich können Wege ausgeschlossen werden, die explizit als nicht rollstuhlgerecht markiert sind: "wheelchair": ["no"]. Sehr schmale Wege können ebenfalls ausgeschlossen werden, wenn sie in der Datenbank als solche markiert sind.

Für Fußgänger-Routing könnte man Privatwege ausschließen: "access": ["private", "no", "customers"]. Oder man schließt Fahrzeugwege aus: "highway": ["motorway", "trunk", "primary"] je nach Anwendungsfall.

Für sichere Routen könnte man unbeleuchtete Wege bei Nacht ausschließen: "lit": ["no"], oder gefährliche Bereiche: "hazard": ["yes"].

Balance zwischen Ausschlüssen und Verfügbarkeit:

Zu viele oder zu strikte Ausschlusskriterien können dazu führen, dass keine Route mehr gefunden wird, insbesondere in dünn vernetzten Graphen. Wenn Start und Ziel nur über ausgeschlossene Wege erreichbar sind, gibt der Algorithmus ein leeres Ergebnis zurück.

Eine Strategie ist, zunächst mit minimalen Ausschlüssen zu arbeiten (z.B. nur Treppen ausschließen) und dann bei Bedarf weitere Kriterien hinzuzufügen. Man kann auch mehrere Routing-Versuche mit unterschiedlichen Ausschlusskriterien machen: Zuerst mit strengen Kriterien, wenn kein Pfad gefunden wird, dann mit gelockerten Kriterien.

Leere excludes:

Wenn keine Ausschlüsse gewünscht sind, kann einfach eine leere Map übergeben werden: {}. In diesem Fall werden alle Relationships potentiell verwendet, die Gewichtung erfolgt dann ausschließlich über propertyWeights und slopeCategories.

Beispiele für verschiedene Szenarien:

Striktes Rollstuhl-Routing:

```
json

{
  "highway": ["steps", "track", "path"],
  "wheelchair": ["no", "limited"],
  "surface": ["gravel", "unpaved", "dirt", "grass"]
}
```

Moderates Rollstuhl-Routing:

```
json

{
  "highway": ["steps"],
  "wheelchair": ["no"]
}
```

Fußgänger ohne Privat-Zugang:

```
json

{
  "access": ["private", "no"]
}
```

Keine Ausschlüsse (alle Wege erlaubt):

```json
{}
```

Parameter 8: poiWeights

Typ: Map von POI-Typen zu Spezifikations-Maps

Pflichtparameter: Nein (Default: {})

Beschreibung:

Der poiWeights-Parameter ist der komplexeste Parameter und steuert die Integration von Points of Interest (POIs) in die Routenplanung. POIs sind wichtige Einrichtungen entlang der Route wie Toiletten, Sitzbänke, Trinkwasserstellen oder andere Annehmlichkeiten. Dieser Parameter definiert, welche POIs gewünscht sind, wie wichtig sie sind, und welche Verteilungsanforderungen entlang der Route gelten.

Grundkonzept:

Wenn poiWeights leer ist ({}), wird keine POI-Suche durchgeführt. Dies ist der Performance-optimierte Modus. Sobald poiWeights POI-Typen enthält, aktiviert sich das POI-System und sucht an jeder Kreuzung nach zugänglichen POIs.

Struktur einer POI-Spezifikation:

Jeder Eintrag in poiWeights hat als Schlüssel den POI-Typ (z.B. "toilets", "bench", "drinking_water") und als Wert eine Map mit drei möglichen Feldern:

```json
"toilets": {
  "importance": 2.0,
  "maxDistance": 100.0,
  "maxInterval": 1500.0
}
```

Das Feld "importance" (Wichtigkeit):

importance ist eine Zahl, typischerweise zwischen 0.1 und 10.0, die angibt, wie wichtig dieser POI-Typ für den Nutzer ist. Höhere Werte bedeuten, dass das Vorhandensein dieses POI-Typs die Route attraktiver macht. Die Wichtigkeit fließt direkt in die POI-Bonus-Berechnung ein.

Ein Wert von 1.0 ist neutral und Standard. Werte über 1.0 (z.B. 2.0 oder 3.0) bedeuten, dass dieser POI-Typ besonders wichtig ist und die Route stark in Richtung dieser POIs beeinflusst werden soll. Werte unter 1.0 (z.B. 0.5) bedeuten, dass der POI nice-to-have ist, aber nicht stark gewichtet werden soll.

Beispiel: Für eine Person mit Inkontinenz-Problemen könnten Toiletten extrem wichtig sein (importance: 3.0), während Bänke weniger kritisch sind (importance: 1.0).

Der Defaultwert, wenn importance nicht angegeben ist, ist 1.0.

Das Feld "maxDistance" (Maximale Gehentfernung):

maxDistance definiert in Metern, wie weit ein POI maximal vom nächsten Wegpunkt entfernt sein darf, um noch berücksichtigt zu werden. Dies ist wichtig, weil POIs oft nicht direkt am Weg liegen, sondern einen kleinen Umweg erfordern.

Ein Wert von 100.0 bedeutet: Der POI muss innerhalb von 100 Metern Gehentfernung von einer Kreuzung auf der Route erreichbar sein. POIs, die weiter entfernt sind, werden ignoriert, auch wenn sie existieren.

Niedrigere Werte (z.B. 50 Meter) bedeuten, dass nur sehr nah gelegene POIs akzeptiert werden, was restriktiver ist. Höhere Werte (z.B. 200 Meter) akzeptieren auch POIs, die einen größeren Umweg erfordern.

Die maxDistance wird verglichen mit der "walk_distance"-Property der ACCESSIBLE_FROM-Beziehung, die den tatsächlichen Gehweg vom POI zur Kreuzung angibt.

Der Defaultwert, wenn maxDistance nicht angegeben ist, ist 100.0 Meter.

Das Feld "maxInterval" (Maximaler POI-Abstand):

maxInterval definiert in Metern, wie weit POIs dieses Typs maximal auseinander liegen dürfen entlang der Route. Dies ist entscheidend für die Routenqualität-Bewertung.

Ein Wert von 1500.0 bedeutet: Entlang der gesamten Route sollten POIs dieses Typs im Abstand von höchstens 1500 Metern vorhanden sein. Für eine 4500 Meter lange Route wären also mindestens 3 POIs erforderlich (bei 0m, 1500m, 3000m, 4500m).

Wenn maxInterval auf 0.0 gesetzt ist oder nicht angegeben ist, bedeutet dies: Keine Verteilungs-Anforderung. Der POI-Typ wird für den Bonus berücksichtigt, aber es wird nicht geprüft, ob genügend POIs vorhanden oder gut verteilt sind. Dies ist nützlich für "nice-to-have" POIs.

maxInterval ist das Herzstück des Routing-Qualitätssystems. Der Algorithmus berechnet, wie viele POIs für die Routenlänge erforderlich wären: erforderliche_POIs = Aufrunden(RoutenLänge / maxInterval). Dann vergleicht er dies mit den tatsächlich gefundenen POIs.

POI-Erkennung im Graphen:

Damit das POI-System funktioniert, müssen POIs im Neo4j-Graphen mit einer speziellen Struktur vorliegen. POIs sind separate Knoten (typischerweise mit Label "POI"), die folgende Properties haben:

"type": Der POI-Typ als String (z.B. "toilets", "bench"). Dies muss exakt mit den Schlüsseln in poiWeights übereinstimmen.

"osm_id": Eine eindeutige ID, typischerweise aus OpenStreetMap, um Duplikate zu erkennen.

optional "location": Geografische Koordinaten als Neo4j Point, falls benötigt.

POIs sind mit Kreuzungen (Intersection-Knoten) durch ACCESSIBLE_FROM-Beziehungen verbunden. Wichtig: Die Richtung ist vom POI zur Kreuzung, also (poi)-[:ACCESSIBLE_FROM]->(intersection).

Die ACCESSIBLE_FROM-Beziehung muss eine Property "walk_distance" haben, die die tatsächliche Gehentfernung in Metern angibt.

Beispiel in Cypher:

```cypher
CREATE (poi:POI {
  type: 'toilets',
  osm_id: '123456789',
  location: point({latitude: 51.5074, longitude: 7.2180})
})

CREATE (poi)-[:ACCESSIBLE_FROM {
  walk_distance: 45.0
}]->(intersection)
```

POI-Bonus-Berechnung im Detail:

Wenn der Algorithmus eine Kreuzung untersucht, sucht er alle POIs, die von dieser Kreuzung erreichbar sind (eingehende ACCESSIBLE_FROM-Beziehungen). Für jeden gefundenen POI wird ein Bonus berechnet, der die Kosten des aktuellen Segments reduziert.

Der Bonus setzt sich aus zwei Komponenten zusammen:

Distanz-Bonus: Je näher der POI an der Kreuzung liegt, desto höher der Bonus. Die Formel lautet: distanz_bonus = max(0, (50 - walk_distance) / 50). Bei 0 Meter Entfernung ist der Bonus 1.0 (maximal). Bei 50 Meter Entfernung ist der Bonus 0.0. Bei mehr als 50 Meter Entfernung ist der Bonus 0.0. Dies bedeutet: Sehr nahe POIs sind deutlich wertvoller als weiter entfernte.

Typ-Bonus: Dies ist einfach die importance aus der poiWeights-Spezifikation für diesen POI-Typ.

Der POI-Bonus für einen einzelnen POI ist: poi_bonus = distanz_bonus × typ_bonus.

Alle POI-Boni an einer Kreuzung werden summiert und dann mit 10.0 multipliziert, um sie in die richtige Größenordnung zu bringen: gesamt_bonus = sum(alle_poi_boni) × 10.0.

Wichtig: Der Gesamtbonus wird auf maximal 30 Prozent der Segmentkosten begrenzt, um extreme Routenverzerrungen zu vermeiden. Die Formel lautet: finaler_bonus = min(gesamt_bonus, segment_kosten × 0.3).

Beispielrechnung:

Angenommen, ein Segment hat Kosten von 200 Metern (nach Steigung und Oberfläche gewichtet). An der Kreuzung werden folgende POIs gefunden:

Toilette bei 20 Meter Entfernung mit importance 2.0: distanz_bonus = (50-20)/50 = 0.6, typ_bonus = 2.0, poi_bonus = 0.6 × 2.0 = 1.2.

Bank bei 10 Meter Entfernung mit importance 1.0: distanz_bonus = (50-10)/50 = 0.8, typ_bonus = 1.0, poi_bonus = 0.8 × 1.0 = 0.8.

Gesamt: (1.2 + 0.8) × 10 = 20.0.

Begrenzung: min(20.0, 200 × 0.3) = min(20.0, 60.0) = 20.0.

Finale Segmentkosten: 200 - 20 = 180 Meter.

Dieses Segment ist also durch die POIs um 10 Prozent attraktiver geworden.

Auswirkung auf die Routenwahl:

Der POI-Bonus bewirkt, dass der Algorithmus leicht zu Routen tendiert, die an POIs vorbeiführen. Je höher die importance und je mehr POIs entlang einer Route, desto attraktiver wird diese Route. Dies kann dazu führen, dass eine etwas längere Route gewählt wird, wenn diese deutlich mehr POIs bietet.

Die Begrenzung auf 30 Prozent stellt sicher, dass POIs nicht zu extremen Umwegen führen. Eine Route, die doppelt so lang ist, wird nicht allein wegen POIs gewählt, auch wenn sie viele POIs hat.

Duplikat-Vermeidung:

Wenn ein POI von mehreren Kreuzungen aus erreichbar ist, könnte er mehrfach gezählt werden. Der Algorithmus verhindert dies durch Duplikat-Erkennung anhand der osm_id. Jeder POI wird nur einmal im Pfad berücksichtigt, auch wenn er mehrfach gefunden wird.

Beispiel-Konfigurationen:

Für eine Person mit Mobilitätsproblemen, die häufig Pausen braucht:

```
json
```

```
{
  "bench": {
    "importance": 2.5,
    "maxDistance": 50.0,
    "maxInterval": 500.0
  },
  "toilets": {
    "importance": 3.0,
    "maxDistance": 100.0,
    "maxInterval": 2000.0
  }
}
```

Für eine längere Wanderung:

```
json

{
  "toilets": {
    "importance": 2.0,
    "maxDistance": 150.0,
    "maxInterval": 3000.0
  },
  "drinking_water": {
    "importance": 1.5,
    "maxDistance": 100.0,
    "maxInterval": 2000.0
  },
  "bench": {
    "importance": 1.0,
    "maxDistance": 50.0,
    "maxInterval": 0.0
  }
}
```

Keine POI-Anforderungen (Performance-optimiert):

```
json

{}
```

Kostenberechnung im Detail

Die Kostenberechnung ist das Herzstück des Algorithmus. Für jedes Straßensegment (Relationship) werden die

Kosten nach folgender Gesamt-Formel berechnet:

Segment-Kosten = Länge × Steigungsgewicht × Oberflächengewicht - POI-Bonus

Diese Formel wird Schritt für Schritt angewendet:

Schritt 1: Basislänge ermitteln

Zunächst wird die physische Länge des Segments aus der "length"-Property der Relationship gelesen. Wenn diese Property fehlt, wird der Defaultwert 1.0 verwendet. Die Länge sollte in Metern angegeben sein für Konsistenz, kann aber prinzipiell jede Einheit haben, solange alle Segmente die gleiche Einheit verwenden.

Schritt 2: Steigungsgewicht bestimmen

Die "slope"-Property der Relationship gibt die Steigung in Prozent an. Positive Werte bedeuten bergauf, negative bergab. Der Absolutwert der Steigung wird verwendet, um die passende Kategorie aus slopeCategories zu finden. Dann wird basierend auf dem Vorzeichen entweder upWeight oder downWeight verwendet. Wenn keine slope-Property vorhanden ist, wird 0.0 angenommen (flach), was zum niedrigsten Gewicht führt.

Schritt 3: Oberflächengewicht berechnen

Für jede Property in propertyWeights wird geprüft, ob das Segment diese Property hat. Wenn ja, wird der Property-Wert als Schlüssel verwendet, um das entsprechende Gewicht nachzuschlagen. Alle gefundenen Gewichte werden miteinander multipliziert. Properties, die nicht vorhanden sind oder deren Werte nicht definiert sind, tragen den Faktor 1.0 bei (neutral).

Schritt 4: Basis-Kosten berechnen

Die Basis-Kosten sind: Länge × Steigungsgewicht × Oberflächengewicht. Diese Zahl repräsentiert die "effektiven Meter", die dieses Segment kostet unter Berücksichtigung aller erschwerenden Faktoren.

Schritt 5: POI-Bonus ermitteln

An der Ziel-Kreuzung des Segments wird nach POIs gesucht. Für jeden gefundenen POI wird ein Bonus berechnet basierend auf Entfernung und Wichtigkeit. Alle POI-Boni werden summiert, mit 10 multipliziert und auf 30 Prozent der Basis-Kosten begrenzt.

Schritt 6: Finale Kosten

Die finalen Segment-Kosten sind: Basis-Kosten minus POI-Bonus. Diese Zahl wird als gScore-Inkrementzum vorherigen Knoten verwendet.

Vollständiges Rechenbeispiel:

Gegeben sei ein Segment mit folgenden Eigenschaften:

length: 180 Meter
slope: 5.5 Prozent (bergauf)

surface: "paving_stones"
smoothness: "intermediate"

An der Zielkreuzung: 1 Toilette bei 35 Meter, 1 Bank bei 15 Meter

Konfiguration:

slopeCategories: maxSlope 8.0 mit upWeight 1.6
propertyWeights: surface paving_stones mit 1.3, smoothness intermediate mit 1.4
poiWeights: toilets mit importance 2.0, bench mit importance 1.5

Berechnung:

Länge: 180 Meter

Steigungsgewicht: 5.5 Prozent liegt unter 8.0 Prozent, positiv, also upWeight 1.6

Oberflächengewicht: 1.3 (paving_stones) × 1.4 (intermediate) = 1.82

Basis-Kosten: 180 × 1.6 × 1.82 = 524.16 Meter

POI-Bonus Toilette: distanz_bonus = (50-35)/50 = 0.3, poi_bonus = 0.3 × 2.0 = 0.6
POI-Bonus Bank: distanz_bonus = (50-15)/50 = 0.7, poi_bonus = 0.7 × 1.5 = 1.05
Gesamt-POI-Bonus: (0.6 + 1.05) × 10 = 16.5
Begrenzung: min(16.5, 524.16 × 0.3) = min(16.5, 157.25) = 16.5

Finale Kosten: 524.16 - 16.5 = 507.66 Meter

Dieses Segment kostet also effektiv 507.66 Meter, obwohl es physisch nur 180 Meter lang ist. Die Steigung und die Oberfläche haben es fast verdreifacht, die POIs haben es um 16.5 Meter günstiger gemacht.

POI-System und Routing-Qualität

Das POI-System verwendet einen mehrstufigen Ansatz zur Bewertung, ob eine gefundene Route die POI-Anforderungen erfüllt. Dies führt zu drei möglichen Qualitätsstufen.

Stufe 1: Perfekte Verteilung (routeQuality = "perfect")

Eine Route erhält die Bewertung "perfect", wenn eine von zwei Bedingungen erfüllt ist:

Entweder es gibt keine POI-Anforderungen (poiWeights ist leer), dann ist jede Route perfekt.

Oder für jeden POI-Typ mit maxInterval größer 0 sind POIs perfekt entlang der Route verteilt. "Perfekt verteilt" bedeutet: Die POIs liegen so, dass der Abstand zwischen aufeinanderfolgenden POIs des gleichen Typs nie größer als maxInterval ist.

Beispiel: Route ist 4000 Meter lang, maxInterval für Toiletten ist 1500 Meter. Perfekt wäre: Toiletten bei Position 0m, 1500m, 3000m und 4000m (oder ähnlich verteilt). Der größte Abstand zwischen zwei aufeinanderfolgenden Toiletten beträgt maximal 1500 Meter.

Bei perfekter Qualität gibt es keine Warnungen, und die Route kann bedenkenlos verwendet werden.

Stufe 2: Ausreichende Anzahl (routeQuality = "good")

Wenn die perfekte Verteilung nicht erreicht wird, prüft der Algorithmus als Fallback, ob zumindest die Anzahl der POIs stimmt. Die erforderliche Anzahl wird berechnet als: erforderliche_POIs = Aufrunden(RouteLänge / maxInterval).

Beispiel: Route ist 3800 Meter, maxInterval ist 1500 Meter. Erforderlich: Aufrunden(3800 / 1500) = Aufrunden(2.53) = 3 Toiletten.

Wenn mindestens diese Anzahl POIs gefunden wurde, erhält die Route die Bewertung "good". Das bedeutet: Es gibt genug POIs, aber sie sind möglicherweise geclustert. Zum Beispiel könnten alle 3 Toiletten im ersten Kilometer liegen, während die letzten 2800 Meter toilettenfrei sind.

Bei "good"-Qualität werden Warnungen generiert: "POI spacing not optimal but sufficient amenities found" sowie für jeden POI-Typ die Information, wie viele gefunden wurden mit dem Hinweis "(may be clustered)".

Stufe 3: Fallback (routeQuality = "fallback")

Wenn weder perfekte Verteilung noch ausreichende Anzahl erreicht wird, gibt der Algorithmus trotzdem die beste gefundene Route zurück, markiert aber die Qualität als "fallback". Dies bedeutet: Die POI-Anforderungen sind nicht erfüllt, aber dies ist die beste Route unter Berücksichtigung aller anderen Faktoren (Steigung, Oberfläche).

Bei "fallback"-Qualität werden detaillierte Warnungen generiert, die auflisten, welche POI-Typen fehlen oder zu wenige sind. Beispiele: "No toilets found (needed: 3)", "Only 1 bench(s) found (needed: 5)".

Der Benutzer muss dann entscheiden, ob er diese Route trotzdem akzeptiert oder seine Anforderungen anpassen möchte (z.B. maxInterval erhöhen oder diesen POI-Typ ganz entfernen).

Interpretation der Qualitätsstufen:

"perfect" bedeutet: Route ohne Einschränkungen nutzbar.

"good" bedeutet: Route nutzbar mit der Einschränkung, dass POIs möglicherweise nicht ideal verteilt sind. Für kurze Routen oder wenn POIs nicht kritisch sind, ist dies meist akzeptabel.

"fallback" bedeutet: Benutzer sollte informiert werden, dass Anforderungen nicht erfüllt sind. Je nach Kritikalität der POIs muss der Benutzer entscheiden oder die Parameter anpassen.

Strategien bei unzureichender Qualität:

Wenn häufig "fallback" auftritt, gibt es mehrere Möglichkeiten: POI-Dichte im Graphen erhöhen (mehr POIs hinzufügen), maxInterval erhöhen (weniger strenge Anforderungen), maxDistance erhöhen (POIs akzeptieren, die weiter vom Weg entfernt sind), importance reduzieren (POIs weniger stark gewichten), oder bestimmte POI-Typen ganz aus den Anforderungen entfernen.

Praktische Beispielkonfigurationen

Im Folgenden sind drei vollständige Konfigurationen für typische Anwendungsfälle dargestellt.

Konfiguration 1: Striktes Rollstuhl-Routing

Diese Konfiguration ist für elektrische Rollstühle optimiert und sehr restriktiv bei Steigungen und Oberflächen.

slopeCategories:

- name "flach", maxSlope 2.0, upWeight 1.0, downWeight 1.05
- name "leicht", maxSlope 4.0, upWeight 1.8, downWeight 1.2
- name "grenzwertig", maxSlope 6.0, upWeight 4.0, downWeight 1.8
- name "zu_steil", maxSlope 999.0, upWeight 50.0, downWeight 5.0

Diese Konfiguration bevorzugt stark flache Bereiche. Alles über 6 Prozent wird praktisch verboten durch extrem hohes Gewicht. Auch bergab wird bei Rollstühlen stärker gewichtet als bei Fußgängern, da Bremsen erforderlich ist.

propertyWeights für surface:

- asphalt 1.0, concrete 1.0 (ideal)
- paved 1.05 (fast ideal)
- paving_stones 1.8 (deutlich schwieriger)
- sett 3.0, cobblestone 5.0 (sehr schwierig)
- compacted 4.0
- gravel 15.0, unpaved 20.0 (praktisch unpassierbar)

propertyWeights für smoothness:

- excellent 0.95, good 1.0
- intermediate 1.8
- bad 5.0, very_bad 15.0

Die Oberflächengewichte sind sehr differenziert. Selbst leichte Abweichungen vom Ideal führen zu merklich höheren Kosten.

excludes:

- highway: steps, track, path (Treppen und Feldwege ausschließen)
- wheelchair: no, limited
- surface: grass, sand (zusätzliche Ausschlüsse bei unmöglichen Oberflächen)

poiWeights:

- toilets: importance 3.0, maxDistance 75.0, maxInterval 2000.0 (sehr wichtig, eng getaktet)
- bench: importance 1.5, maxDistance 50.0, maxInterval 1000.0 (Pausen wichtig)

Diese Konfiguration würde für eine Person im Elektro-Rollstuhl sehr sichere und komfortable Routen liefern, allerdings mit längeren Wegen und möglicherweise weniger direkten Verbindungen.

Konfiguration 2: Fußgänger mit Gehbehinderung

Diese Konfiguration ist weniger restriktiv als Rollstuhl-Routing, berücksichtigt aber dennoch, dass Steigungen und schlechte Oberflächen problematisch sind.

slopeCategories:

- name "flach", maxSlope 5.0, upWeight 1.0, downWeight 0.95
- name "moderat", maxSlope 10.0, upWeight 1.6, downWeight 1.05
- name "steil", maxSlope 15.0, upWeight 3.0, downWeight 1.3
- name "sehr_steil", maxSlope 999.0, upWeight 8.0, downWeight 2.0

Steigungen bis 5 Prozent sind kein großes Problem. Bis 10 Prozent ist es schwieriger aber machbar. Alles über 15 Prozent wird stark vermieden.

propertyWeights für surface:

- asphalt 1.0, concrete 1.0, paved 1.05
- paving_stones 1.2, compacted 1.4
- sett 1.8, cobblestone 2.5
- gravel 3.0, unpaved 5.0

propertyWeights für smoothness:

- excellent 0.9, good 1.0, intermediate 1.3
- bad 2.5, very_bad 6.0

Die Gewichte sind moderater als bei Rollstuhl-Routing. Auch schlechtere Oberflächen sind noch gangbar, nur mit höherem Aufwand.

excludes:

- highway: steps (nur Treppen sind absolut tabu)

poiWeights:

- bench: importance 2.5, maxDistance 50.0, maxInterval 500.0 (häufige Pausen wichtig)
- toilets: importance 2.0, maxDistance 100.0, maxInterval 1500.0

Diese Konfiguration ist flexibler und findet eher direkte Routen, berücksichtigt aber trotzdem die besonderen Bedürfnisse.

Konfiguration 3: Standard-Fußgänger (Performance-optimiert)

Diese Konfiguration ist für normale Fußgänger ohne besondere Einschränkungen und optimiert auf schnelle Berechnung.

slopeCategories:

- name "alle", maxSlope 999.0, upWeight 1.05, downWeight 0.98

Alle Steigungen werden fast gleich behandelt, mit minimalem Aufschlag für bergauf.

propertyWeights:

- surface: asphalt 1.0, paved 1.0, paving_stones 1.05 (nur große Unterschiede zählen)

excludes: (leer)

poiWeights: (leer)

Diese minimalistische Konfiguration führt zu sehr schnellen Berechnungen und findet im Wesentlichen die kürzesten Wege mit minimalen Anpassungen für Steigungen.

Performance und Fehlerbehandlung

Performance-Charakteristik:

Die Performance des Algorithmus hängt von mehreren Faktoren ab. Die wichtigsten sind: Anzahl der Knoten im Graphen, Dichte der Vernetzung, Länge der gesuchten Route, Anzahl der POI-Typen mit maxInterval-Anforderungen.

Typische Laufzeiten: Kurze Routen unter 1 Kilometer mit wenigen hundert Knoten: 20 bis 50 Millisekunden. Mittlere Routen 1 bis 5 Kilometer mit 500 bis 2000 Knoten: 50 bis 200 Millisekunden. Lange Routen über 5 Kilometer mit mehr als 2000 Knoten: 200 Millisekunden bis 2 Sekunden. Sehr komplexe Szenarien in dichten urbanen Netzwerken: über 2 Sekunden möglich.

Die POI-Suche kann zusätzliche Zeit kosten, insbesondere wenn viele POI-Typen definiert sind und der Graph viele POIs enthält. Jede Kreuzung muss nach POIs durchsucht werden, was bei Tausenden von Kreuzungen signifikant wird.

Optimierungsstrategien:

Um die Performance zu verbessern, können folgende Maßnahmen ergriffen werden:

Indizes: Erstellen Sie Indizes auf häufig abgefragte Properties: CREATE INDEX intersection_location FOR (n:Intersection) ON (n.x, n.y) und CREATE INDEX poi_type FOR (p:POI) ON (p.type).

Minimale Steigungskategorien: Weniger Kategorien bedeuten schnellere Lookups. Drei bis vier Kategorien sind meist ausreichend.

Beschränkte propertyWeights: Definieren Sie nur Properties, die wirklich relevant sind. Jede zusätzliche Property erfordert zusätzliche Lookups.

Leere poiWeights wenn möglich: Wenn keine POIs benötigt werden, lassen Sie poiWeights leer. Dies überspringt die gesamte POI-Suche-Logik.

Graph-Preprocessing: Stellen Sie sicher, dass der Graph gut strukturiert ist. Vermeiden Sie zu viele kleine Segmente, konsolidieren Sie wo möglich.

Fehlerbehandlung:

Leeres Ergebnis: Wenn die Prozedur Stream.empty() zurückgibt, wurde kein Pfad gefunden. Mögliche Ursachen sind: Start und Ziel sind nicht verbunden, zu restriktive excludes blockieren alle Pfade, Steigungsgewichte sind so extrem, dass alle Pfade als "unendlich teuer" gelten. Debug-Strategie: Prüfen Sie mit einfachem shortestPath, ob überhaupt eine Verbindung existiert. Lockern Sie dann schrittweise die excludes und Gewichte.

Ungültige Element-IDs: Wenn eine Element-ID nicht existiert, wird eine NotFoundException geworfen. Stellen Sie sicher, dass die IDs korrekt aus dem Graphen extrahiert wurden mit elementId(node).

Fehlende Properties: Der Algorithmus ist robust gegen fehlende Properties und verwendet Defaultwerte (0.0 für numerische Properties, 1.0 für Gewichte). Allerdings führen fehlende Koordinaten (xProp, yProp) zu schlechter Heuristik-Performance.

Parsing-Fehler: Wenn Properties nicht-numerische Werte haben, die nicht geparst werden können, werden Defaultwerte verwendet. Dies sollte nicht zu Fehlern führen, kann aber zu unerwarteten Gewichtungen führen.

Graph-Validierung:

Vor dem produktiven Einsatz sollte der Graph validiert werden: Prüfen Sie, dass alle Intersection-Knoten xProp und yProp haben. Prüfen Sie, dass alle Relationships eine length-Property haben. Validieren Sie die ACCESSIBLE_FROM-Beziehungen für POIs. Stellen Sie sicher, dass osm_id bei POIs eindeutig ist. Testen Sie mit verschiedenen Start-Ziel-Kombinationen, ob Routen gefunden werden.

Ende der Dokumentation

Diese detaillierte Dokumentation sollte alle Aspekte des Algorithmus abdecken und als umfassendes Nachschlagewerk dienen. Bei weiteren Fragen oder für Hilfe bei der Konfiguration wenden Sie sich bitte an das Entwicklungsteam.