TECHNISCHE UNIVERSITÄT
CHEMNITZ

Fakultät für Informatik

Automatische Generierung von Navigationsgraphen auf Basis von OpenStreetMap-Innenraumkarten

Forschungsseminar

Vorgelegt von:	Bettina Auschra
Betreuer:	Dipl.-Ing. Thomas Graichen, Dr. Andreas Müller
Vorgelegt am:	21.02.2018
Studiengang:	Informatik für Geistes- und Sozialwissenschaftler

Inhaltsverzeichnis

Forschungsseminar: Automatische Generierung von Navigationsgraphen auf Basis von OpenStreetMap-Innenraumkarten

1. Einleitung

Ziel dieses Forschungsseminars ist es, ein Programm zur automatischen Generierung von Navigationsgraphen für OpenStreetMap-Innenraumkarten zu entwickeln. Im Folgenden wird zunächst eine allgemeine Übersicht über die bisherige Forschung zum Thema der automatischen Generierung von Navigationsgraphen gegeben. Anschließend wird einer der vorgestellten Ansätze in Python implementiert und das Ergebnis ausgewertet werden.

2. Überblick zur aktuellen Forschung

Eingegangen wird auf Navigationsmodelle, die sich auch auf der Grundlage von OpenStreetMap-Daten anwenden lassen. Der Ausgangspunkt sind also Modelle, in denen die Positionen von Räumen, Korridoren und Türen bereits bekannt sind, da dies bei OpenStreetMap-Daten auch der Fall ist.

Ein Ansatz ist die Berechnung von Knotenpunkten durch Triangulierung. Dabei werden die Punkte innerhalb eines Polygons, zum Beispiel gebildet aus den begehbaren Teilen eines Gebäudes, so miteinander verbunden, dass eine minimale Anzahl an Dreiecken gebildet wird (s. de Berg et al., S. 46). Anschließend wird der duale Graph gebildet. Dabei dienen die Schwerpunkte der Dreiecke als Knotenpunkte des Graphen. Die Schwerpunkte der Dreiecke, die sich eine Seite teilen, werden mit einer Kante verbunden (de Berg et al., S. 47). Der Nachteil bei dieser Methode ist, dass der dabei entstandene Graph teilweise umständliche Routen liefert (s. Abb. 1).

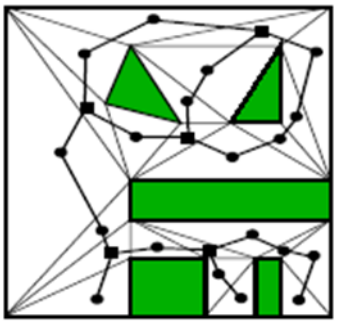


Abb. 1 Demyen & Buro, S. 943: Triangulierung und dualer Graph

Liu und Zlatanova sowie Yuan und Schneider schlagen Navigationsmodelle vor, bei denen die Kanten der Graphen, falls möglich, von einer Tür direkt zur nächsten verlaufen (Liu & Zlatanova, S. 1; Yuan & Schneider, S. 4). Ist eine direkte Verbindung zwischen zwei Türen nicht möglich, wird der kürzestmögliche Pfad, der über die konvexen Punkte des Raum- bzw. Korridorpolygons verläuft, bestimmt (s. Liu & Zlatanova, S. 3; Yuan & Schneider, S. 8).

Liu und Zlatanova stellen ihren Ansatz Navigationsmodellen gegenüber, bei denen in Korridoren Mittellinien gebildet werden (s. Abb. 2). Die Autoren bevorzugen ihren eigenen Ansatz, weil sie die Auffassung vertreten, dass die dabei entstehenden Wege der Art, wie Fußgänger wirklich laufen, entsprechen (s. Liu & Zlatanova, S. 1).

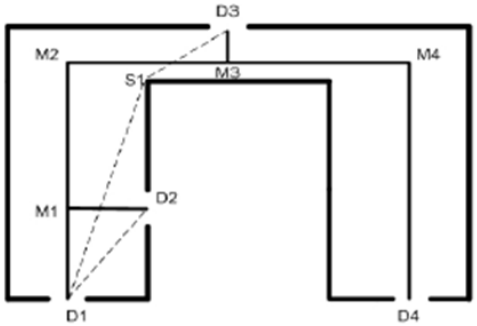


Abb. 2 (Liu & Zlatanova, S. 2): Vergleich zwischen Graphkanten, die von Tür zu Tür bzw. auf Mittellinien verlaufen

Yuan und Schneider kritisieren des Weiteren, dass Modelle, bei denen repräsentative Punkte, wie zum Beispiel der Raummittelpunkt, als Knotenpunkte gewählt werden, nicht die kürzeste Verbindung und damit auch nicht die tatsächlich von Fußgängern zurückgelegte Entfernung zwischen zwei Punkten anzeigen (s. Yuan & Schneider, S. 4).

Als Beispiel für ein Navigationsmodell, in dem Mittellinien und repräsentative Punkte eine Rolle spielen, ist der Ansatz von Worboys und Yang zu nennen.

Ausgehend von einem Gebäudeplan, den die Autoren als *Structure graph* bezeichnen und in dem Portale, Wände und Sackgassen eingezeichnet sind (s. Abb. 3, oben links), bilden sie zunächst eine so genannte *Dual map* (Worboys & Yang, S. 5). Wie beim *Dual graph* der Triangulierung gibt es bei der *Dual map* für jedes Polygon, das einen Raum oder Korridor darstellt, einen repräsentativen Punkt[1]. Die repräsentativen Punkte derjenigen Raum- bzw. Korridorpolygone, die eine Kante (Wand) miteinander teilen, werden miteinander verbunden (s. Abb. 3, oben rechts).

Anschließend entfernen Worboys und Yang alle Knoten aus der *Dual map*, die aus einer Kante des *Structure graph* gebildet wurden, auf der sich kein Portal befindet (Worboys & Yang, S. 6). Den nun erhaltenen Graphen nennen die Autoren *Simplified dual map* (s. Abb. 3, unten rechts).

Als Letztes fügen sie jedes Portal als neuen Knoten zwischen den zwei Knoten der *Simplified dual map* ein, die die Räume repräsentieren, auf deren gemeinsamer Kante des *Structure Graphs* sich das jeweilige Portal befindet (Worboys & Yang, S. 6). Sie nennen den damit erhaltenen Graph *Skeleton navigation graph* (s. Abb. 3, unten links).

[1] Bei der Triangulierung handelt es sich bei den repräsentativen Punkten um die Schwerpunkte der Dreiecke.

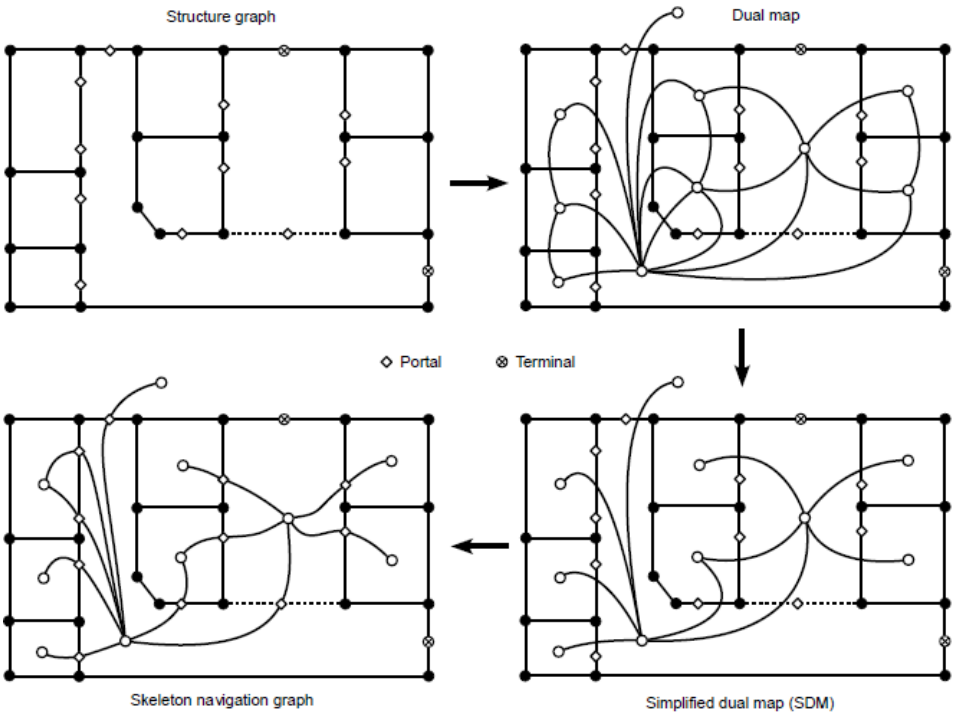


Abb. 3 (Worboys & Yang, S. 7): Vom Structure graph *zum* Skeleton navigation graph

Den *Skeleton navigation graph* ergänzen Worboys und Yang um weitere Knoten. Sie unterscheiden dabei zwischen einfachen Räumen, einfachen rechteckigen Korridoren und komplexen Korridoren und Räumen (Worboys & Yang, S. 9-10).

Bei einfachen konvexen Räumen verschieben Worboys und Yang den aktuellen Knoten des Raumes in den Schwerpunkt des Raumes (Worboys & Yang, S. 9)[2].

Bei einfachen rechteckigen Korridoren berechnen sie zunächst eine Mittellinie, die parallel zu den Korridorseiten und senkrecht zu den Korridorenden verläuft (Worboys & Yang, S. 10). Im Fall von Türen, die auf den Korridorseiten liegen, werden Knotenpunkte auf der Mittellinie gegenüber der Türen platziert. Gegenüber von Türen, die an den Korridorenden liegen, werden Knotenpunkte im Abstand von der Hälfte der Korridorbreite hinzugefügt (Worboys & Yang, S. 10).

Handelt es sich um einen komplexen Raum, überprüfen Worboys und Yang als Erstes, ob es innerhalb des Polygons, das den Raum darstellt, einen Kern gibt, also einen Bereich innerhalb des Raums, von dem aus der gesamte Raum einzusehen ist. Falls dies der Fall ist, fügen sie einen Knoten in den Schwerpunkt des Kerns hinzu. Anderenfalls unterteilen sie den Raum in

[2] Worboys und Yang erhalten die repräsentativen Punkte eines Raumes zunächst mit Hilfe einer *Combinatorial Map*, die den Ausgangspunkt ihrer Arbeit bildet. Da diese Punkte aber verschoben werden (s. Worboys & Yang, S. 9) und *Combinatorial Maps* in dieser Arbeit keine Rolle spielen, wird hierauf nicht eingegangen.

mehrere Teilpolygone und berechnen dort jeweils den Kern und den Schwerpunkt, dessen Position sie anschließend als Knoten speichern. (Worboys & Yang, S. 11f.)

Ein weiterer Ansatz zur Bildung von Navigationsgraphen ist der des *Straight Skeleton*. Das *Straight Skeleton* kann man sich wie ein Dach vorstellen, das die Grundfläche eines Polygons überspannt. Es wird konstruiert, indem man die Grundfläche des Polygons schrittweise verkleinert (Haunert & Sester, S. 4).

Hierbei wird zwischen zwei Arten von Ereignissen unterschieden – *edge events* und *split events*. Ein *edge event* tritt auf, wenn eine Kante gelöscht wird, weil die zwei angrenzenden Kanten aufeinander treffen (Haunert & Sester, S.4; s. Abb. 4, links).

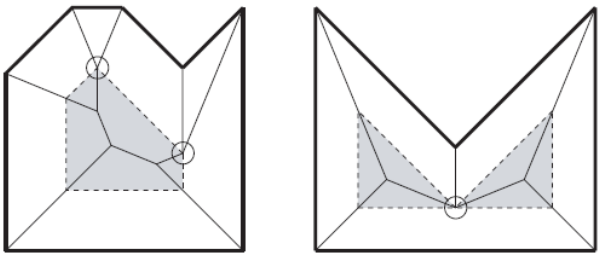


Abb. 4 (Haunert & Sester, S. 4): links: zwei edge events; *rechts:* split event

Bei einem *split event* entstehen zwei neue Polygone, weil eine Kante auf einen Punkt trifft, der mit zwei anderen Kanten verbunden ist (Haunert & Sester, S. 4; s. Abb. 4, rechts).

Straight Skeletons eignen sich besonders gut, um Mittellinien zu berechnen (s. Haunert & Sester, S. 23). Im Rahmen des Forschungsseminars wird dieser Ansatz den Ansätzen von Liu und Zlatanova sowie Yuan und Schneider vorgezogen, weil es in dieser Arbeit nicht darum geht, den kürzesten Weg zu finden, sondern möglichst übersichtlich mögliche Wege darzustellen. Es wird die Auffassung vertreten, dass ein Navigationsgraph, der sich auf die Konstruktion von Mittellinien stützt, verständlicher und übersichtlicher für den Benutzer ist. Dies wird auch durch die Untersuchung von Worboys und Yang gestützt, die Probanden Übersichtspläne und dort verlaufende Routen zeichnen ließen (Worboys & Yang, S.17).

Da der Umfang des Forschungsseminars für eine Implementierung des Ansatzes von Worboys und Yang nicht ausreichen würde und es für die Berechnung eines *Straight Skeleton* bereits frei zugängliche Implementationen gibt, wurde sich entschieden, mit Letzterem zu arbeiten.

3. Anwendung des *Straight Skeleton* auf OpenStreetMap-Daten

3.1. Polyskel

In dieser Arbeit wird als Grundlage die bereits bestehende *Straight-Skeleton*-Implementierung *polyskel*[3] in Python verwendet. *Polyskel* wird in der Version des GitHub-Benutzers *yonghah*

[3] s. https://github.com/yonghah/polyskel

verwendet, da diese in Python 3.5 programmiert wurde, im Gegensatz zum Original des GitHub-Benutzers *botffy*[4], das über Python 2.7 läuft.

Laut Aussagen des Autors *botffy* basiert die Implementierung des *Straight Skeleton* auf dem Konferenzbeitrag „Straight skeleton implementation“ von Petr Felkel und Štěpán Obdržálek[5]. Die genaue Funktionsweise und Bewertung des Ansatzes von Felkel und Obdržálek ist kein Bestandteil dieses Forschungsseminars. Ebenso wenig wird überprüft, inwieweit *botffy* die Beschreibung der Autoren umgesetzt hat.

Die Berechnung des *Straight Skeleton* erfolgt über die Funktion *skeletonize* in der Datei „polyskel.py“. Der Funktion werden die Punkte eines Polygons in einer Liste übergeben. Die Koordinaten der Punkte werden als Tupel übergeben, die aus einem x- und einem y-Wert bestehen. Hierbei ist es wichtig, dass die Reihenfolge der Koordinaten gegen den Uhrzeigersinn[6] erfolgt und es keine negativen Werte geben darf. Optional können der Funktion auch zusätzlich die Koordinaten von Löchern innerhalb des Polygons übergeben werden. Diese Funktion wird jedoch in diesem Forschungsseminar nicht gebraucht.

Als Ergebnis liefert die Funktion eine Liste von sogenannten *Subtrees* zurück[7]. Diese *Subtrees* enthalten jeweils einen Ausgangspunkt (*source*) und eine aus einem oder mehreren Verbindungspunkten bestehende Liste (*sinks*). Verbindet man jeden Ausgangspunkt mit den dazugehörigen Verbindungspunkten, erhält man das *Straight Skeleton* (s. Abb. 5).

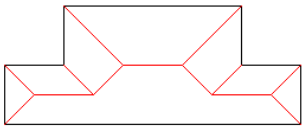


Abb. 5: Mit polyskel *berechnetes Straight Skeleton*

Im Folgenden werden die einzelnen Funktionen, die für die Generierung von Wegen auf Grundlage von OpenStreetMap-Daten geschrieben wurden, beschrieben. Die Funktionen wurden mit zwei OpenStreetMap-Testdateien getestet (s. Anhang).

3.2. Einlesen der Daten und Extrahieren der Türen und Räume

Mit Hilfe der Funktion *readData* werden die Koordinaten von Türen und Räumen in dem Gebäude, für das Wege generiert werden sollen, bestimmt. Der Funktion werden die bekannten OpenStreetMap-Daten in einer Datei übergeben. In dieser Datei wurden zuvor alle Längen- und

[4] s. https://github.com/botffy/polyskel

[5] zu finden unter http://www.dma.fi.upm.es/personal/mabellanas/tfcs/skeleton/html/documentacion/Straight-%20Skeletons%20Implementation.pdf

[6] *botffy* geht davon aus, dass sich der Punkt (0|0) in der linken oberen Ecke des Koordinatensystems befindet.

[7] Beispiel für einen *Subtree*: Subtree(*source*=Point2(75.00, 125.00), *height*=75.0, *sinks*=[Point2(0.00, 50.00), Point2(0.00, 200.00)]) (Die angegebene Höhe ist für diese Zwecke irrelevant.)

Breitengrade in metrische Werte umgerechnet. Aus diesem Grund werden im Folgenden die Werte der Breitengrade als x-Werte und die Werte der Längengrade als y-Werte bezeichnet.

Da OpenStreetMap-Daten im XML-Format gespeichert werden, wird mit Hilfe der Klasse *ElementTree* aus der Python-internen XML-Bibliothek zunächst die gesamte Datei geparst.

Es wird davon ausgegangen, dass die Daten in der gegebenen OpenStreetMap-Datei so strukturiert sind, dass Türen entweder als *node* oder als *way* gespeichert werden und einen *tag* mit dem Schlüssel „entrance“ oder „door“ enthalten. Ziel ist es, für jede Tür einen aus x- und y-Koordinaten bestehenden Punkt zu erhalten.

Zunächst werden aus der geparsten Datei alle *nodes* und *ways* mit den entsprechenden Schlüsselwörtern extrahiert. Für den Fall, dass die Türen als *ways* gespeichert wurden, wird mit Hilfe einer weiteren Funktion der Schwerpunkt des Polygons berechnet, das aus den x- und y-Werten der einzelnen *nodes* des *ways* gebildet wird. Die Koordinaten der berechneten Schwerpunkte bzw. der Türen, die direkt als *nodes* gespeichert wurden, werden nach Etagen geordnet in einer Liste gespeichert.

Räume und Korridore werden gemäß dem *Simple Indoor Tagging*[8] als *ways* gekennzeichnet, die das Schlüssel-Wert-Paar „indoor=room“ bzw. „indoor=corridor“ als *tag* besitzen. Jeder *way* wird deshalb zusätzlich daraufhin überprüft, ob er einen *tag* mit den Werten „room“ oder „corridor“ besitzt. Die Koordinaten der *nodes*, aus denen die *ways* der gefunden Räume[9] gebildet sind, werden nach Räumen getrennt in Listen gespeichert und ebenfalls der entsprechenden Etage untergeordnet.

Die Listen der Räume und Türen werden als Ergebnis zurückgeliefert.

3.3. Kombination von Türen und Räumen

Mit der Funktion *addDoors* wird überprüft, welche Türen in welche Räume führen. Die Funktion erhält als Parameter die Liste der Räume und die Liste der Türen. Die Liste der Koordinaten der einzelnen Raumpolygone soll so aktualisiert werden, dass am Ende die Koordinaten der Türen ebenfalls in den dazugehörigen Räumen gespeichert sind.

[8] s. https://wiki.openstreetmap.org/wiki/Simple_Indoor_Tagging (Für Türen gibt es noch keine einheitliche Regelung, deshalb erfolgt der Verweis erst an dieser Stelle.)

[9] Da in diesem Forschungsseminar die Navigationsgraphen für Räume und Korridore auf dieselbe Art berechnet werden, wird im Folgenden nur noch von Räumen gesprochen.

Dafür wird jeder Raum einzeln betrachtet. Alle Punkte des aktuellen Raumpolygons, die aufeinander folgen, werden paarweise nacheinander betrachtet[10]. Durch jedes Punktepaar wird als Erstes eine Gerade gezogen. Anschließend wird für jede Tür, die sich auf derselben Etage wie der aktuelle Raum befindet, eine weitere Gerade bestimmt, die durch die Position der Tür verläuft und die erste Gerade senkrecht schneidet.

Beträgt der Abstand zwischen dem Schnittpunkt der Gerade und der Tür weniger als 0,3m und befindet sich der Schnittpunkt auf der Strecke zwischen den beiden Ausgangspunkten, wird davon ausgegangen, dass die Tür zu dem Raum gehört (s. Abb. 6)[11]. Die Koordinaten der betrachteten Tür werden dementsprechend in die Koordinatenliste des aktuellen Raums zwischen den beiden Ausgangspunkten eingefügt.

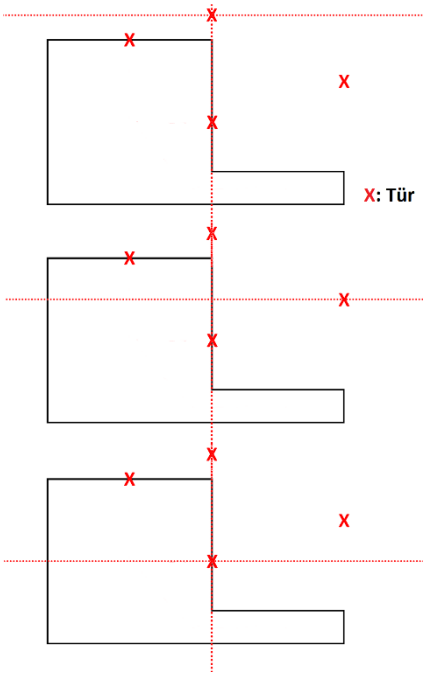


Abb. 6: Funktionsweise von addDoors*: nur die unterste Tür erfüllt beide Kriterien und wird zwischen den beiden Ausgangspunkten in die Liste der Punkte des Raums eingefügt*

3.4. Generierung von Wegen mit Hilfe von *polyskel*

In der Funktion *findWays* soll mit Hilfe der Funktion *skeletonize* von *polyskel* für jeden einzelnen Raum ein *Straight Skeleton* berechnet werden.

Wie bereits erwähnt, muss die Reihenfolge der Punkte des Polygons, das man der Funktion *skeletonize* übergibt, gegen den Uhrzeigersinn erfolgen. Aus diesem Grund wird jedes gespeicherte Raumpolygon daraufhin überprüft, ob dieses Kriterium gegeben ist. Falls nötig, wird die Reihenfolge der *nodes* in der Liste des Raums umgekehrt. Anschließend wird die Koordinatenliste des aktuellen Raumes an *skeletonize* übergeben.

[10] Die Punkte des Polygons sind bereits geordnet, da dies eine Voraussetzung für die Syntax von *ways* in OpenStreetMap ist.

[11] Türen, die sich in der Mitte eines Raumes, der durch eine Trennwand o.ä. geteilt ist, befinden, werden hiermit nicht erfasst.

Wie oben beschrieben, gibt die Funktion *skeletonize* als Ergebnis eine Liste von *Subtrees* zurück, die einen Ausgangspunkt und einen oder mehrere Verbindungspunkte enthält. Befinden sich die Koordinaten eines Ausgangs- oder Verbindungspunktes in der Liste der Koordinaten des aktuellen Raums, aber nicht in der Liste der Türen, handelt es sich um eine Verbindung, die ignoriert werden kann, da ein Weg in die Ecken des Raumes nicht sinnvoll ist (s. Abb. 7). Anderenfalls wird die Verbindung zwischen den Koordinaten des Ausgangspunktes und den Koordinaten des Verbindungspunktes als zukünftiger *way* gespeichert.

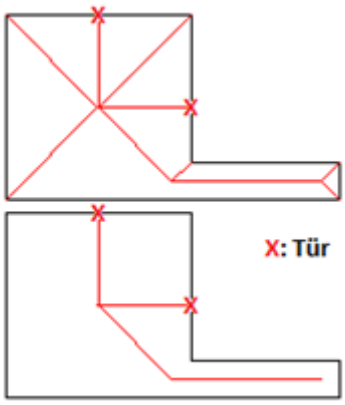


Abb. 7 oben: Straight Skeleton*; unten:* Straight Skeleton *nach Entfernung unnötiger Verbindungen*

Jedem Punkt eines gefunden Weges wird eine eindeutige ID zugeordnet. Diese IDs werden an späterer Stelle benötigt, um die extrahierten Wege wieder ins OpenStreetMap-Format zu schreiben.

Die gefundenen Wege werden nach Räumen getrennt gespeichert.

3.5. Kurze Wege zu längeren zusammenfassen

Da die derzeit gefunden Wege jeweils nur aus zwei Knoten bestehen, wurde die Funktion *longWays* geschrieben, in der die bereits gefundenen Wege zu längeren Wegen zusammengefasst werden. Die Funktion erhält als Parameter die Liste der Wege und die Liste der Türen.

In dieser Arbeit wird zwischen *Decision nodes* und *Terminal nodes* unterschieden. *Decision nodes* werden definiert als Knoten eines Graphen, die mehr als zwei direkte Verbindungen zu anderen Knoten besitzen. *Terminal nodes* sind Knoten, die nur eine direkte Verbindung zu einem anderen Knoten haben.

Die Wege, die diese Funktion zurückliefert, verlaufen so, dass jeder Weg mit einem *Terminal node*, einem *Decision node* oder einer Tür beginnt und auch mit einem *Terminal no*de, einem *Decision node* oder einer Tür endet. Zwischen einem Anfangs- und einem Endpunkt darf es keinen *Decision node* und auch keine Tür geben. Es kann dabei durchaus vorkommen, dass es auch weiterhin Wege gibt, die aus lediglich zwei Knoten bestehen.

Da die Wege innerhalb der Wegeliste nach Räumen getrennt gespeichert wurden, werden jeder Raum und die sich darin befindenden Wege einzeln betrachtet. Die zwei *nodes* des ersten *ways*, der noch nicht in einem neu angelegten *way* zu finden ist, werden in eine neue Liste kopiert.

Anschließend wird die Liste der verbliebenen *ways* des Raumes, die sich noch nicht in einem neuen *way* befinden, daraufhin untersucht, ob sich die Koordinaten eines der beiden Knoten eines Weges an der ersten oder letzten Stelle der neu angelegten Liste befinden. Stimmen die Koordinaten eines Knotens mit denen des ersten Knotens in der neuen Liste überein, wird der andere Knoten des betrachteten Weges der Liste als erstes Element hinzugefügt. Stimmen die Koordinaten eines Knotens mit denen des letzten Knotens in der neuen Liste überein, wird der andere Knoten des Weges dem Ende der Liste angehängt.

Dieser Vorgang wird so lange wiederholt, bis der neu angelegten Liste keine Knoten mehr hinzugefügt werden können. In diesem Fall wird die Liste als neuer *way* mit dem dazugehörigen Wert der Etage gespeichert. Je nachdem, ob bereits alle *ways* des Raumes abgearbeitet wurden, wird der nächste Raum betrachtet oder der Vorgang mit dem nächsten unbearbeiteten Weg des aktuellen Raumes wiederholt.

Die dabei entstehenden Wege hängen von der Reihenfolge, in der die kurzen Wege vorher gespeichert wurden, ab, da immer der erste passende Knoten in den neuen Weg eingefügt wird (s. Abb. 8).

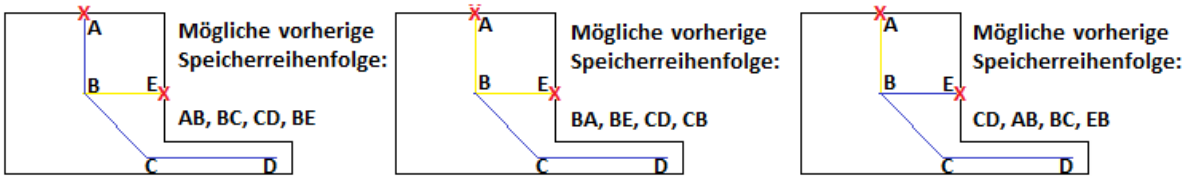


Abb. 8: Verschiedene Wege, die aus denselben Ausgangswegen bei unterschiedlicher vorheriger Speicherreihenfolge entstehen

Nachdem alle Wege in allen Räumen betrachtet wurden, hat man eine Liste mit neuen Wegen erhalten, in der jedoch noch Wege vorkommen können, die mehr als zwei *Decision nodes*[12] enthalten. Um dies zu ändern, werden zunächst die *Decision nodes* bestimmt.

Anschließend werden alle gefunden Wege, die mehr als zwei Knoten besitzen, daraufhin überprüft, ob sich ein *Decision node* an einer anderen Stelle als der ersten oder letzten befindet. In diesem Fall wird ein neuer Weg gespeichert, der aus dem gefundenen *Decision node*, allen darauf folgenden Punkten und der entsprechenden Etage besteht. Mit Ausnahme des *Decision node* werden alle Knoten, die sich im neu angelegten Weg befinden, aus der Liste des aktuellen Weges entfernt.

[12] Der Fall, dass sich eine Tür in der Mitte eines Weges befindet, kann ausgeschlossen werden, weil die gefundenen Wege bereits nach Räumen getrennt gespeichert sind.

3.6. Wege vereinfachen

Die Funktion *simplifyWays* wurde geschrieben, um angelegte Wege zu vereinfachen. Dabei wurde davon ausgegangen, dass bei der Erstellung eines *Straight Skeleton* Knoten entstehen, die den Weg komplizierter machen und deshalb entfernt werden können (s. Abb. 9).

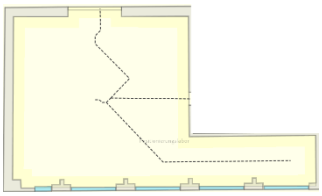


Abb. 9: aus Straight Skeleton *entstandene Wege*

Der Algorithmus funktioniert so, dass jeder Weg einzeln betrachtet wird. Besteht ein Weg aus lediglich zwei Knoten, kann dieser nicht weiter vereinfacht werden.

Anderenfalls werden immer drei aufeinanderfolgende Knoten untersucht. Durch den ersten Knoten und den dritten Knoten wird eine Gerade gezogen. Anschließend wird eine Senkrechte zu dieser Gerade gesucht, die durch den zweiten Knoten geht. Schneidet diese zweite Gerade die erste zwischen dem ersten und dritten Knoten und liegt der Abstand zwischen dem zweiten Knoten und dem Schnittpunkt der beiden Geraden unter einem zuvor festgelegten Schwellenwert, wird der zweite Knoten aus dem Weg entfernt (s. Abb. 10). Dieser Vorgang wird solange wiederholt, bis sich der Weg nicht weiter vereinfachen lässt.

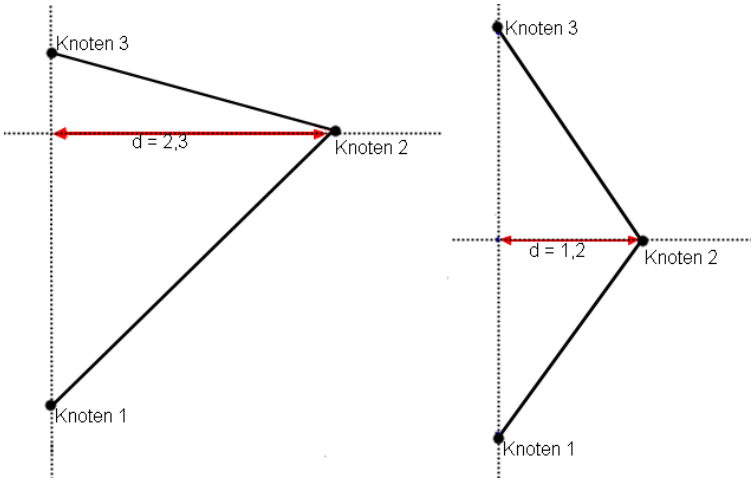


Abb. 10: Bei einem Schwellenwert von 1,5m würde Knoten 2 links nicht entfernt werden, rechts jedoch schon

Die Wahl des Schwellenwertes ist wichtig für das Ergebnis des Algorithmus. Im Fall der ersten OpenStreetMap-Testdatei lieferte ein Schwellenwert von 1,5m das beste Ergebnis (s. Abb. 11, links). Bei einem Schwellenwert von 0,5m gab es nur eine geringe Veränderung (s. Abb. 11, Mitte) und bei einem Schwellenwert von 2,5m gab es ungültige Wege (s. Abb. 11, rechts).

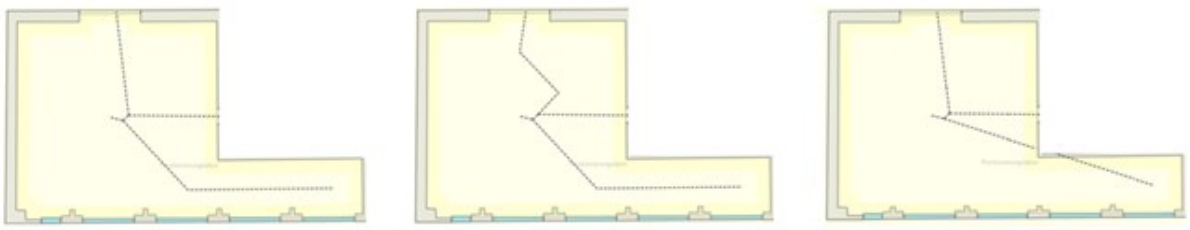


Abb. 11: Vereinfachung der Wege aus Abb. 9 bei verschiedenen Schwellenwerten - links: 1,5m, Mitte: 0,5m, rechts: 2,5m

Bei der Anwendung des Algorithmus auf die zweite Testdatei zeigte sich, dass der Vereinfachungsalgorithmus bereits bei einem Schwellenwert von 1,0m ungültige Wege zurücklieferte und die zuvor berechneten Wege eigentlich keiner Vereinfachung mehr bedurften (s. Anhang).

3.7. Wege im OpenStreetMap-Format speichern

In der Funktion *writeOsm* werden die gefundenen Wege im OpenStreetMap-Format in einer Datei gespeichert. Dabei wird sich an dem im OpenStreetMap-Wiki beschriebenen Format orientiert[13].

Mit Hilfe der bereits verwendeten Klasse *ElementTree* wird zunächst ein Wurzelelement *osm* mit dem dazugehörigen Attributen angelegt. Als Nächstes wird jeder Punkt, der in den zu speichernden Wegen verwendet wird, als Kindknoten *node* von *osm* mit den entsprechenden Koordinaten und IDs als Attributen angelegt. Anschließend wird für jeden gefundenen Weg ein Kindknoten *way* für *osm* erstellt und als Attribut eine neue ID angelegt. Jeder *way* erhält außerdem als Kindknoten eine Referenz *nd* für jeden Knoten des aktuell betrachteten Weges und die *tags*, die den *way* als einen Weg in einem Innenraum auf einer bestimmten Etage kennzeichnen. Als Letztes wird die erstellte XML-Baumstruktur in eine neue Datei geschrieben.

4. Fazit und Ausblick

Nach Betrachtung verschiedener Ansätze für die automatische Generierung von Navigationsgraphen wurde sich für den Ansatz des *Straight Skeleton* entschieden, um die automatische Generierung von Navigationsgraphen auf der Basis von OpenStreetMap-Innenraumkarten durchzuführen. Es wurde eine Möglichkeit entwickelt, die OpenStreetMap-Daten auszulesen und so zu speichern, dass sie an die bestehende *Straight-Skeleton*-Implementierung *polyskel* übergeben werden können. Die dabei entstandenen Wege wurden mit einer weiteren Funktion vereinfacht. Die Wahl des Schwellenwerts in der Vereinfachungsfunktion ist dabei maßgeblich für das Ergebnis der Vereinfachung.

[13] s. https://wiki.openstreetmap.org/wiki/OSM_XML

Dadurch, dass bei der Berechnung des *Straight Skeleton* alle Ecken des Polygons zum entstehenden Graph beitragen, entstehen auch Wege, die lediglich zu Ausbuchtungen in der Wand – etwa durch vorhandene Fenster – führen (s. Abb. 12). Dies hängt davon ab, wie genau die Raumgrenzen in OpenStreetMap kartographiert wurden. Eine Möglichkeit wäre, das Raumpolygon vor der Übergabe an die Funktion *skeletonize* von *polyskel* zunächst durch eine weitere Funktion zu vereinfachen[14].

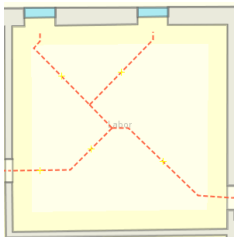


Abb. 12: Wege, die zu Fenstern führen

Bei der Anwendung des Algorithmus auf die zweite OpenStreetMap-Testdatei zeigte sich, dass der größte Raum nicht gefunden wurde, weil dieser statt des *tag* „room“ nur den *tag* „area“ besaß (s. Anhang).

Die anderen Räume in der zweiten Testdatei waren so kartographiert worden, dass das entstandene Raumpolygon auch Teile der Wand umschloss. Dies führte dementsprechend auch zu inkorrekten Wegen über Wände (s. Abb. 13).

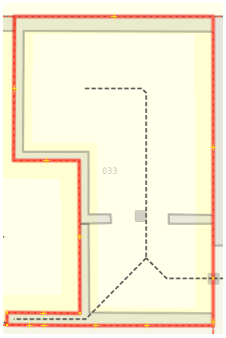


Abb. 13: rot: Grenzen des Raumpolygons; grau: Wände

Beide Probleme ließen sich durch eine konsequentere Kartographierung verhindern.

Da der Umfang des Forschungsseminars nicht für zwei Implementierungen reichte, wurde sich für die Umsetzung des aus Gründen der Übersichtlichkeit favorisierten Ansatzes des *Straight Skeleton* entschieden. Eventuell wäre es aber sinnvoll, in Zukunft zusätzlich zu dem Graphen, der durch den *Straight-Skeleton*-Algorithmus bestimmt wird, auch einen Graphen zu entwickeln, der sich auf den Ansatz von Liu und Zlatanova bzw. Yuan und Schneider stützt. So wäre gewährleistet, dass der Benutzer des Navigationssystems nicht erst in die Mitte des Raumes geführt wird, wenn es eine direkte Verbindung zwischen zwei Türen gibt.

Zukünftig sollte auch auf die Verbindungen zwischen den einzelnen Etagen eingegangen werden. Außerdem sollte untersucht werden, ob es innerhalb eines Raumes unüberwindbare Elemente, wie etwa Säulen, gibt. Da der Funktion *skeletonize* auch Löcher im Polygon übergeben werden können, sollte dies grundsätzlich möglich sein.

[14] Dieses Verfahren wird auch vom GitHub-Benutzer *yonghah* angewandt – siehe https://github.com/yonghah/polyskel/blob/master/Create%20layout%20network%20using%20straight%20skeletons%20.ipynb

Literaturverzeichnis

De Berg, M., Cheong, O., Van Kreveld, M., & Overmars, M. (2008). *Computational geometry*. Berlin, Heidelberg: Springer.

Demyen, D., & Buro, M. (2006). Efficient triangulation-based pathfinding. In *Aaai*. Retrieved from https://vvvvw.aaai.org/Papers/AAAI/2006/AAAI06-148.pdf

Liu, L., & Zlatanova, S. (2011). A "door-to-door" path-finding approach for indoor navigation. *Proceedings Gi4DM 2011: GeoInformation for Disaster Management, Antalya, Turkey, 3-8 May 2011*. Retrieved from http://www.isprs.org/proceedings/2011/Gi4DM-/PDF/OP05.pdf

Haunert, J. H., & Sester, M. (2008). Area collapse and road centerlines based on straight skeletons. Retrieved from https://www.ikg.uni-hannover.de/fileadmin/ikg/staff/publications/-Begutachtete_Zeitschriftenartikel_und_Buchkapitel/HaunertSester_geoinformatica2008.pdf

Worboys, M., & Yang, L. (2015). Generation of navigation graphs for indoor space. *International Journal of Geographical Information Science*, *29*(10). Retrieved from http://gala.gre.ac.uk/13719/1/13719_WORBOYS_Generation_of_navigation_AAM_(2015).pdf

Yuan, W., & Schneider, M. (2010). iNav: An Indoor Navigation Model Supporting Length-Dependent Optimal Routing. Retrieved from https://www.cise.ufl.edu/~mschneid/Research/-papers/YS10AGILE.pdf

Anhang

OpenStreetMap-Testdaten und berechnete Wege

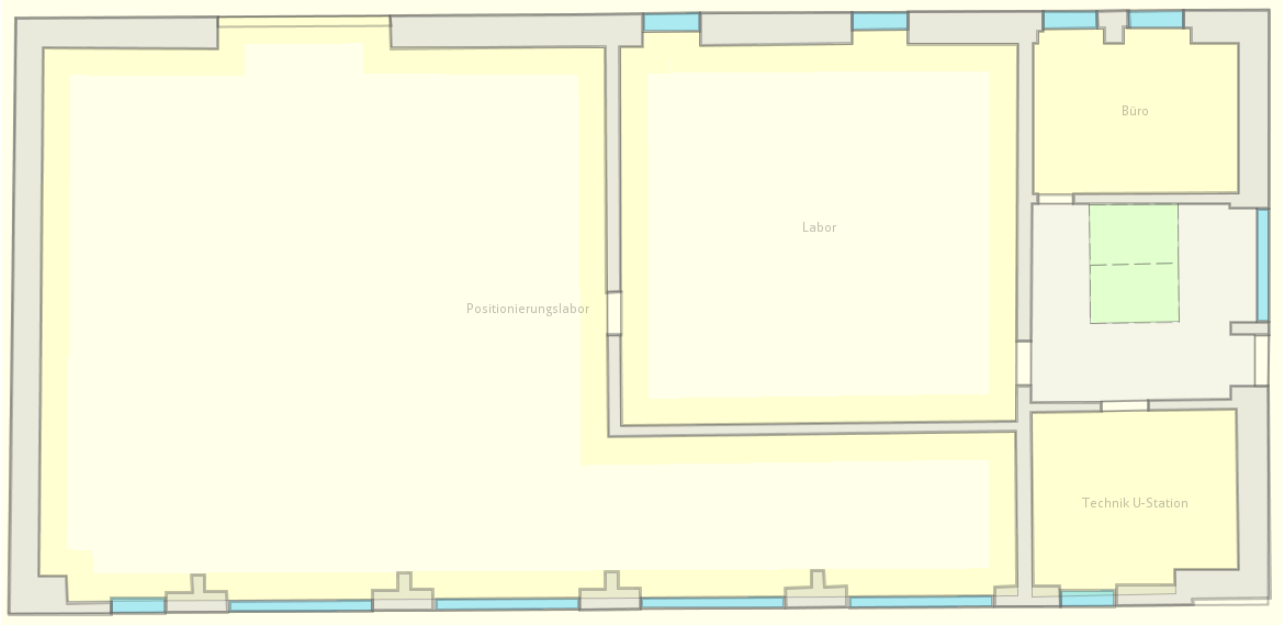
1. Auszug aus der ersten OpenStreetMap-Testdatei

```
<?xml version='1.0' encoding='UTF-8'?>
<osm version='0.6' upload='false' generator='JOSM'>
  <node id='-40422' action='modify' lat='3.80102697149' lon='20.29793924784' />
  <node id='-40432' action='modify' lat='0.30806322062' lon='18.23892726424' />
  <node id='-40442' action='modify' lat='10.58966490459' lon='6.80176449503' />
  <node id='-40464' action='modify' lat='10.39503588056' lon='3.72847184402' />
  <node id='-40552' action='modify' lat='3.64733595418' lon='21.87555131222' />
  <node id='-40644' action='modify' lat='3.60534756865' lon='19.46970764479' />
  <node id='-40662' action='modify' lat='0.78483138441' lon='20.7729917567' />
  <node id='-40680' action='modify' lat='0.36086614598' lon='20.79409878656' />
  <node id='-40684' action='modify' lat='3.63137155817' lon='20.30279018687' />
  <node id='-40702' action='modify' lat='10.41527031299' lon='6.80925015414' />
  <node id='-40772' action='modify' lat='0.80169896392' lon='21.90358277574' />
  <node id='-40870' action='modify' lat='10.57222235905' lon='3.72752977998' />
  <node id='-40890' action='modify' lat='3.58356594892' lon='18.2303878566' />
  <node id='-40904' action='modify' lat='3.77500298469' lon='19.46485670664' />
      (. . .)
  <way id='-40952 ' action='modify'>
    <nd ref='-40772' />
    <nd ref='-40552' />
    <nd ref='-40890' />
    <nd ref='-40432' />
    <nd ref='-40680' />
    <nd ref='-40662' />
    <nd ref='-40772' />
    <tag k='indoor' v='room' />
    <tag k='level' v='0' />
    <tag k='name' v='Technik U-Station' />
  </way>
      (. . .)
  <way id='-40962' action='modify'>
    <nd ref='-40684' />
    <nd ref='-40422' />
    <nd ref='-40904' />
    <nd ref='-40644' />
    <nd ref='-40684' />
    <tag k='access' v='private' />
    <tag k='door' v='yes' />
    <tag k='level' v='0' />
  </way>
  (. . .)
  <way id='-40990' action='modify'>
    <nd ref='-40702' />
    <nd ref='-40442' />
    <nd ref='-40870' />
    <nd ref='-40464' />
    <nd ref='-40702' />
    <tag k='entrance' v='private' />
    <tag k='level' v='0' />
  </way>
  (. . .)
</osm>
```

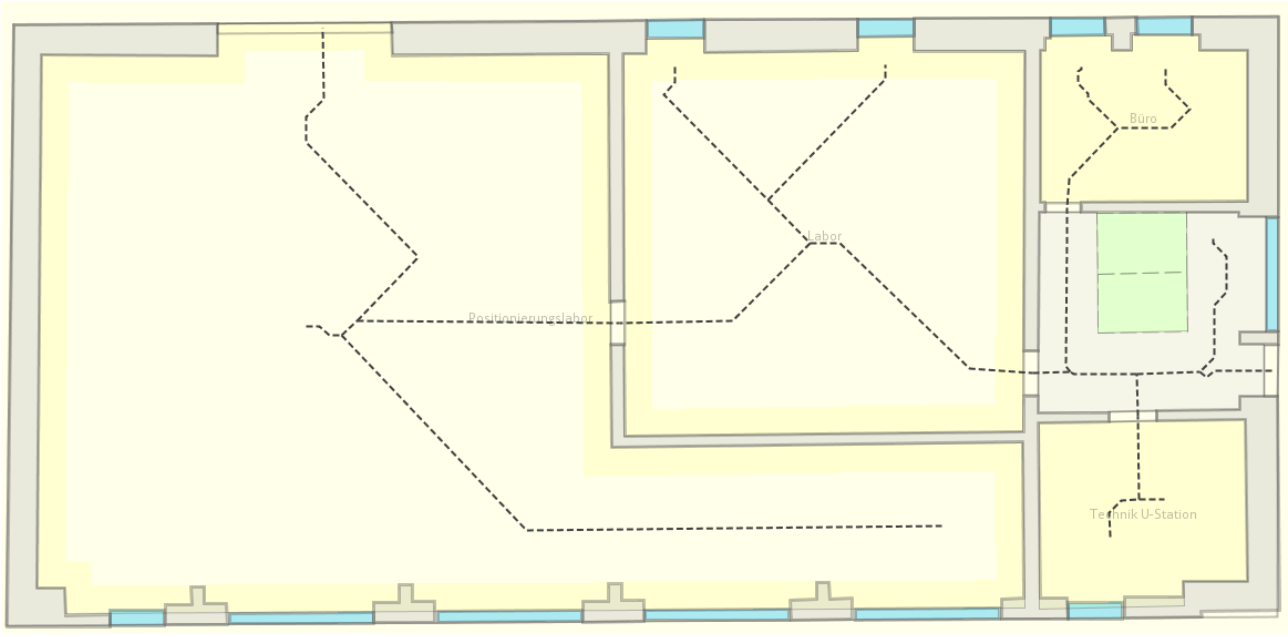
Beispiel für einen Raum

Beispiele für Türen

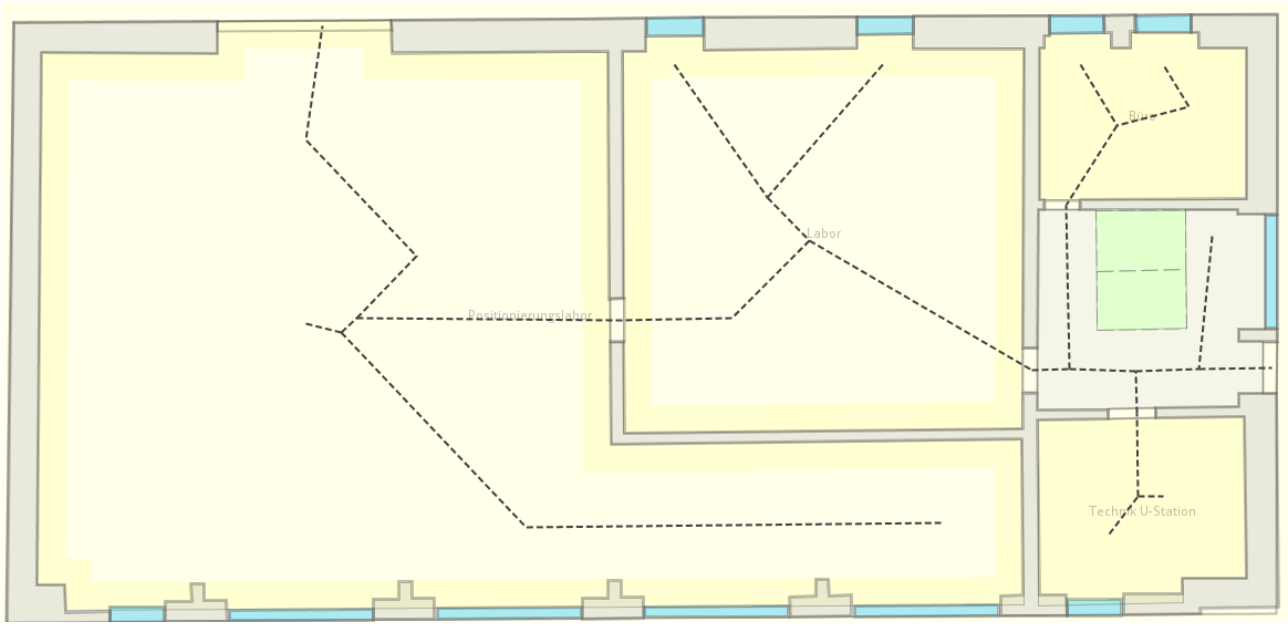
2. Darstellung der ersten OpenStreetMap-Testdatei in JOSM



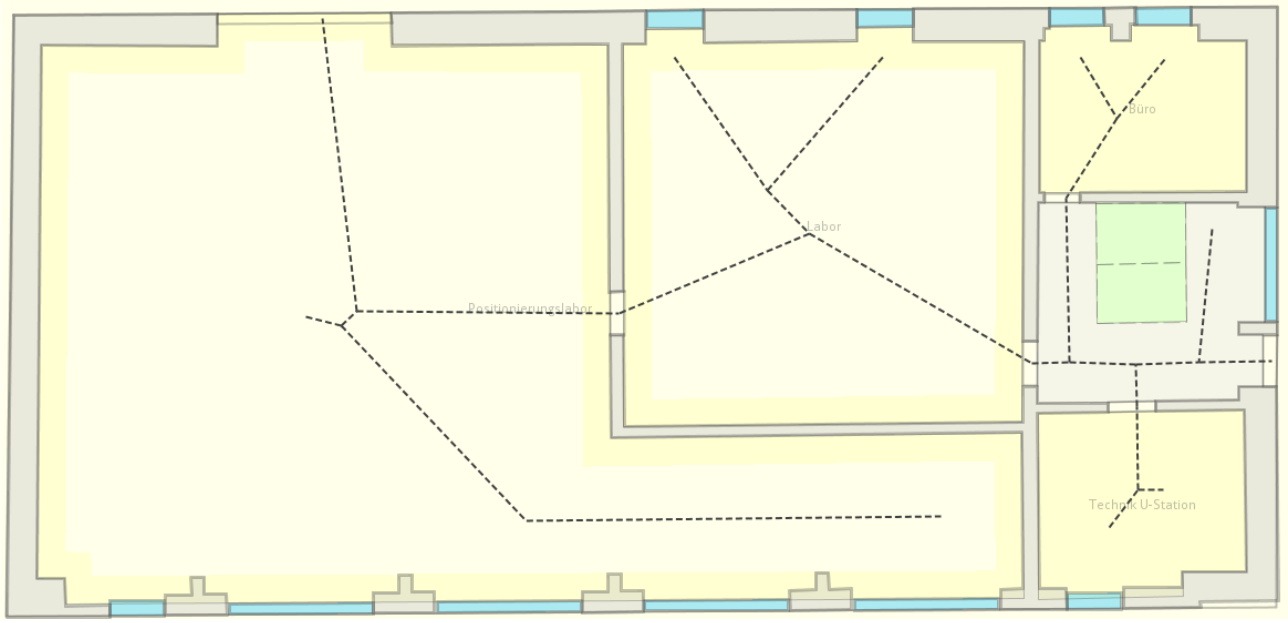
3 Darstellung der berechneten Wege



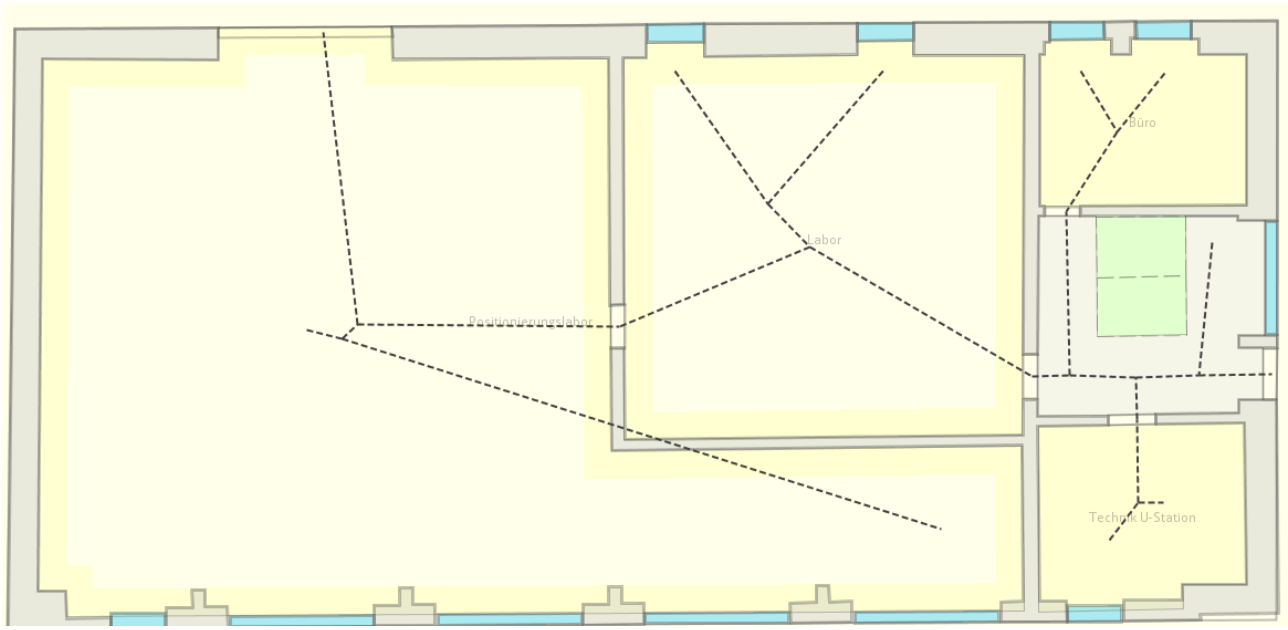
4. Darstellung der berechneten Wege nach Anwendung des Vereinfachungsalgorithmus mit einem Schwellenwert von 0,5m



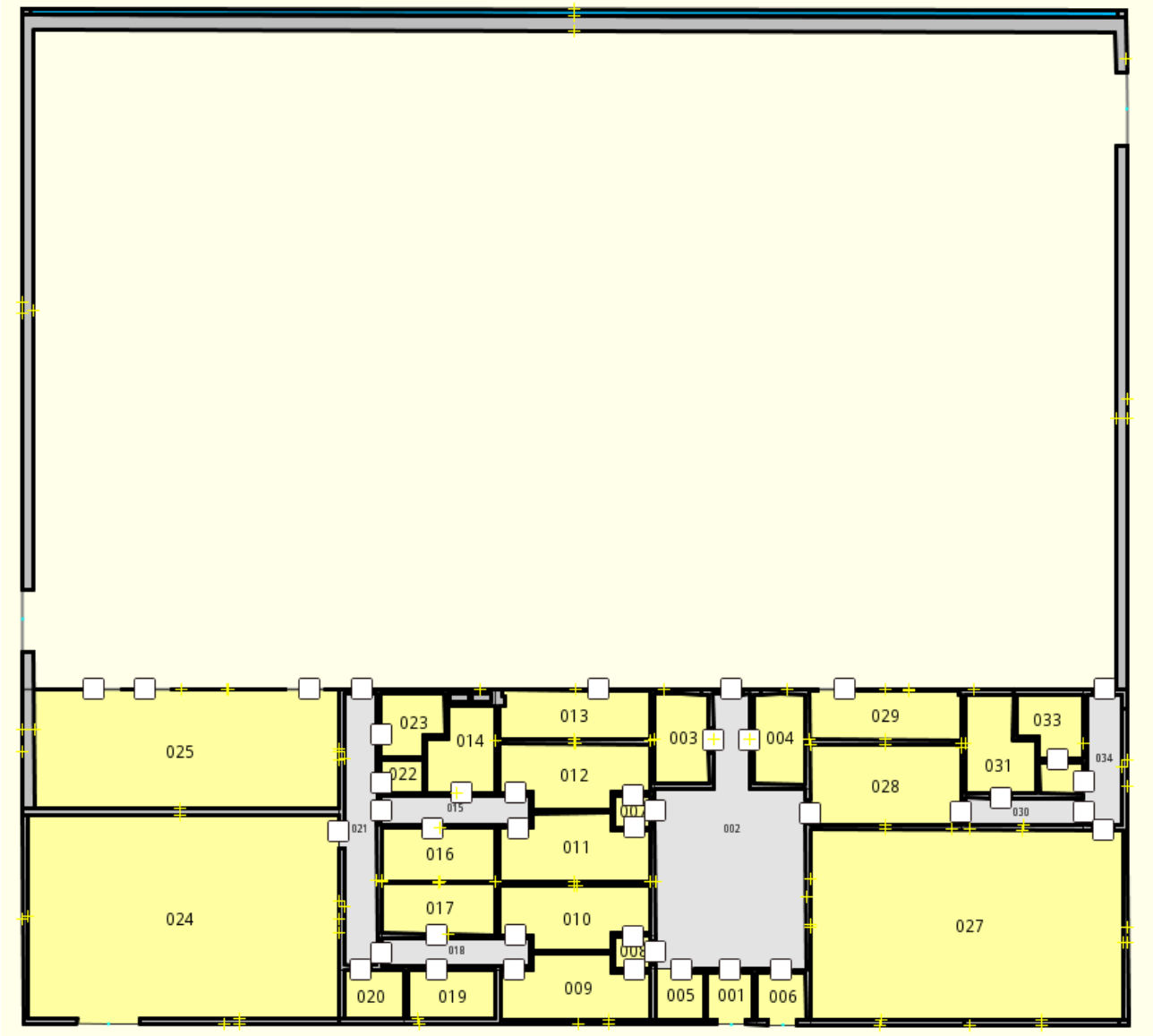
5. Darstellung der berechneten Wege nach Anwendung des Vereinfachungsalgorithmus mit einem Schwellenwert von 1,5m



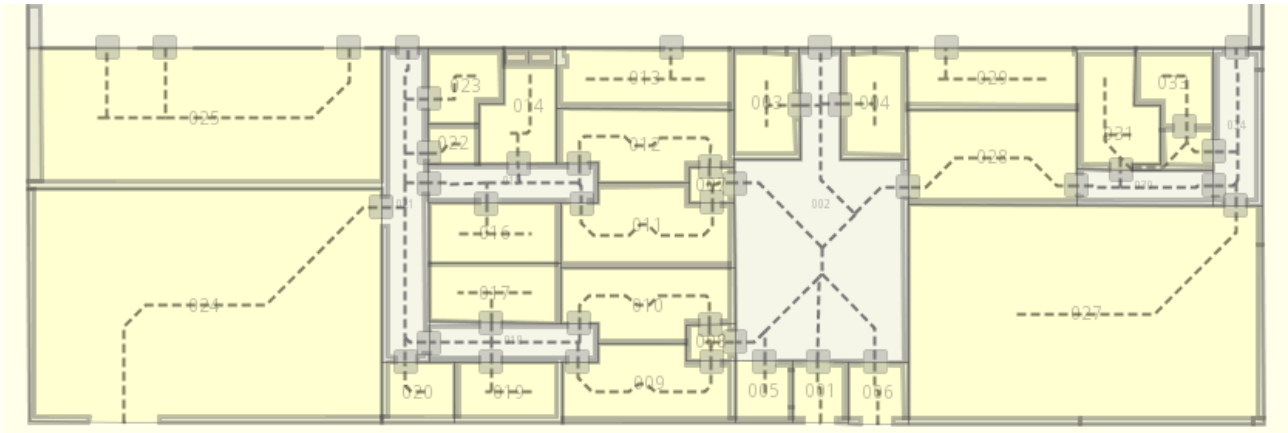
6. Darstellung der berechneten Wege nach Anwendung des Vereinfachungsalgorithmus mit einem Schwellenwert von 2,5m



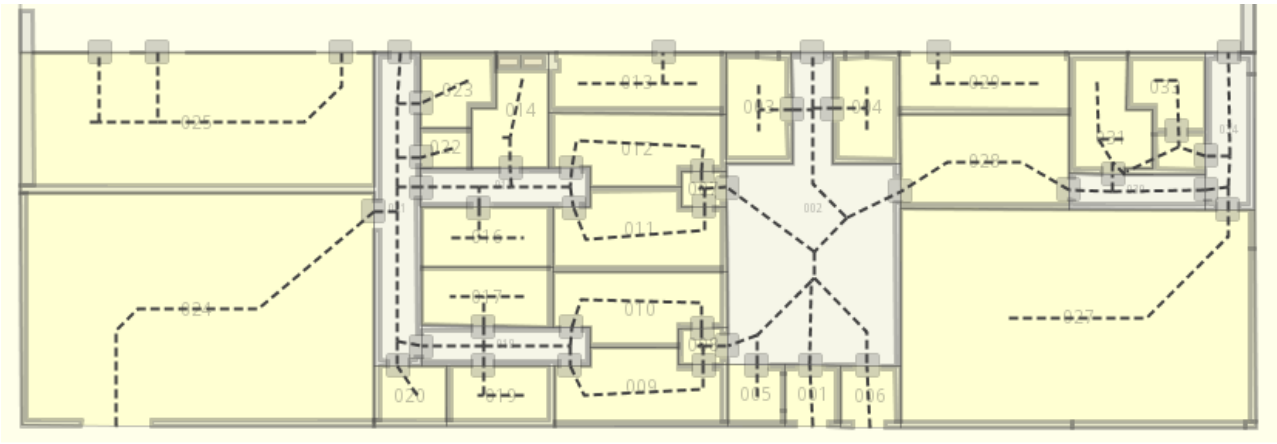
7. Darstellung der zweiten OpenStreetMap-Testdatei in JOSM



8. Darstellung der berechneten Wege



9. Darstellung der berechneten Wege nach Anwendung des Vereinfachungsalgorithmus mit einem Schwellenwert von 0,5m



10. Darstellung der berechneten Wege nach Anwendung des Vereinfachungsalgorithmus mit einem Schwellenwert von 1m