1 Basic Idea

The basic idea is to seperate the gridgraph into a set of pairs $\mathbb{S} = \{(A_i, B_i) \mid i \in \mathbb{N}\}$ such that $\forall (A, B) \in \mathbb{S} : A, B \subseteq V \wedge A \cap B = \emptyset$ and for each pair $u, v \in V$ there exists a pair $(A, B) \in \mathbb{S}$ such that $a \in A \wedge b \in B$ or $b \in A \wedge a \in B$. Furthermore, given a pair $(A, B) \in \mathbb{S}$, it should be easy to calculate optimal distances for paths from nodes $u \in A$ to $v \in B$ and the other way arround. Since for every pair of nodes $u, v \in V$ there exists a pair $(A, B) \in \mathbb{S}$ with $a \in A \wedge b \in B$ or $b \in A \wedge a \in B$, the calculation of the optimal distance between u and v boils down to finding such a pair $(A, B) \in \mathbb{S}$ and then reporting the optimal distance.

This means the challenge of finding the optimal distances for a gridgraph fast breaks down into two subchallenges:

- **Preprocessing:** calculating set $\mathbb{S}$ with the given properties and a way of reporting the optimal distances between nodes of A and B.
- **Query:** for a given pair $u, v \in V$, finding a set $a \in A \wedge b \in B$ or $b \in A \wedge a \in B$ must be fast.

In the following different approaches for the preprocessing and query phase are presented.

2 Separation

The idea of node separations comes from the well-separated pair decomposition, which organizes a given point set P such that all of $\binom{P}{2}$ distances between points are clustered in into a linear amount of clusters, and that all distances in each cluster are roughly the same.

In order to achieve this a QuadTree is build for the point set P and then algorithm 1 with the QuadTree root as input is called to calculate the well separation.

Algorithm 1: SeparateWSPD

Data: QuadTree Nodes X, Y
Result: Cluster Set $\mathbb{S}$
if $X = Y$ **then**
  return $\emptyset$
if $size(X) < size(Y)$ **then**
  return $SeparateWSPD(Y, X)$
if $areWellSeparated(X, Y)$ **then**
  return $\{(X, Y)\}$
return $\bigcup_{C \in child(X)} SeparateWSPD(C, Y)$

Notice how the $areWellSeparated(X, Y)$ function can be used specify when two clusters are well-separated. In order to be useable for shortest path distance queries, three different $areWellSeparated$ function could be used.

1. **Trivial Separation:** X and Y are well separated
   $\iff \forall u \in X : \forall v \in Y : d_{opt}(u, v) = |u_1 - v_1| + |u_2 - v_2|$
   where $d_{opt}(u, v)$ is the optimal distance between u and v.

2. **Portal Separation:** X and Y are well separated
   $\iff \exists u_{portal} \in X : \exists v_{portal} \in Y : \forall u \in X : \forall v \in Y : u_{portal} \in \pi(u, v) \wedge v_{portal} \in \pi(u, v)$
   where $\pi(u, v)$ is one optimal path from u to v.
   This means that there exists a portal node in X and a portal node in Y which is part of all shortest paths from X to Y.

3. **Barrier Separation:**
   not implemented
   X and Y are well separated
   $\iff \exists b \in V : \forall u \in X : \forall v \in Y : b \in \pi(u, v)$
   which means that there exists a barrier node $b \in V$ that every shortest path from X to Y goes throught this node.

2.1 Trivial Separation

To check wether two quadtree nodes X and Y are trivial separatable is easily achievable with algorithm 2. An example of a trivial separation in a gridgraph can be seen in figure 1.

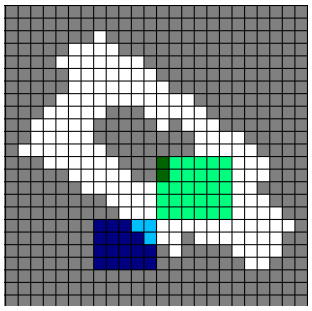

Figure 1: trivial separation example

Query

When X and Y which are trivial separated are known for a given query (x, y) the answer to the query is $|x_1 - y_1| + |x_2 - y_2|$.

Algorithm 2: TrivialSeparationCheck

Data: QuadTree Nodes X, Y
Result: *Boolean*
for $x \in X$ **do**
  for $y \in Y$ **do**
    $dist \longleftarrow d(x, y)$;
    $trivialDist \longleftarrow |x_1 - y_1| + |x_2 - y_2|$;
    if $dist \neq trivialDist$ **then**
      return $FALSE$
return $TRUE$

2.1.1 Lazy Path Reconstruction

not implemented

2.2 Portal Separation

To calculate if two quadtree nodes X and Y are protal separatabel the two portals need to be found first. The two portal candidates of a two quadtree nodes are the two nodes $x \in X$ and $y \in Y$ with the minimal distance.

$$(x_{portal}, y_{portal}) = \underset{(x,y) \in X \times Y}{\operatorname{argmin}} \; d_{opt}(x, y) \tag{1}$$

With those two portal candidates algorithm 3 can find out if two quadtree nodes are portal separated.

Algorithm 3: PortalSeparationCheck

Data: QuadTree Nodes X, Y, Portal Candidates x_{portal}, y_{portal}
Result: *Boolean*
for $x \in X$ **do**
  for $y \in Y$ **do**
    if
    $d_{opt}(x, y) \neq d_{opt}(x, x_{potal}) + d_{opt}(x_{portal}, y_{portal}) + d_{opt}(y, y_{potal})$
    then
      return $FALSE$
return $TRUE$

An example of a portal separation in a gridgraph can be seen in figure 2.

2.2.1 Query

In order to answer a query, for each portal separated quadtree node pair (X, Y), the two portals x_{portal}, y_{portal} and the distance between them $d_{opt}(x_{portal}, y_{portal})$

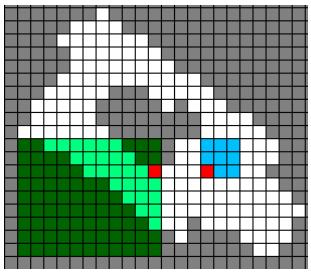

Figure 2: portal separation example

needs to be stored. With those things stored the algorithm 4 can be used to answer queries.

Algorithm 4: PortalQuery

Data: QuadTree Nodes X, Y, Query Nodes $u \in X, v \in Y$, Portal Candidates x_{portal}, y_{portal}
Result: $d_{opt}(u, v)$
if $u = x_{portal} \wedge v = y_{portal}$ **then**
  return $d_{opt}(x_{portal}, y_{portal})$
if $u = x_{portal}$ **then**
  return $d_{opt}(x_{portal}, y_{portal}) + PortalQuery(y_{portal}, v)$
if $v = y_{portal}$ **then**
  return $PortalQuery(x, x_{portal}) + d_{opt}(x_{portal}, y_{portal})$
return
  $PortalQuery(x, x_{portal}) + d_{opt}(x_{portal}, y_{portal}) + PortalQuery(y_{portal}, v)$

2.3 Separation Optimization

2.3.1 Avoiding Neigbours

Since the distance from a node to its neigbour is easly calculatable in a gridgraph, all separations (X, Y) with $|X| = |Y| = 1$ and $\forall x \in X : \forall y \in Y :$ *x and y are neigbours* can be omitted.

2.3.2 Separation Weight Optimization

The algorithm 1 does only compare quadtree nodes with a heigh difference of at most 1. While this leads to optimal and unambiguous separations for points, it does not for nodes in a gridgraph. To further optimize the separations a comparison between different separations is needed. in the following, the weight of a separation (X, Y) is defined as:

$$weight((X,Y)) = |\{x \mid x \in X \ \wedge \ x \ is \ walkable\}| \cdot |\{y \mid y \in Y \ \wedge \ y \ is \ walkable\}|$$

Intuitively the weight of a separation is the number of node pairs for which it is able to answer optimal distance queries.

Furthermore domination of one separation of another is defined as:

$$(X,Y) \preceq (A,B) = (A \subseteq X \wedge B \subseteq Y) \vee (B \subseteq X \wedge A \subseteq Y)$$

If one separation is dominated by another separation it can be omitted, since all node pair queries can also be answered by the dominating one.

Separations can be seen as blue edges between the well-separated nodes in the quadtree. Such a set of blue nodes can be optimized by trying to push the blue edges higher in the quadtree. Since the original algorithm only connects two quadtree nodes with a high difference of at most 1, often such blue edges can be pushed higher in the quadtree. In order to do this the two nodes of a separation are used as a lower bound, which means the blue edge will not be pushed below the nodes of quadtree. As an upper bound the first quadtree node which both nodes of the separation are a subset of is used. Let $parent(A,B)$ be the set of nodes in the quadtree of which are A and B subsets, and $parent(A)$ all nodes which are node A is a subset of. Then in order to push the the blue edge higher, algorithm 5 can be used to push a blue edge higher. When a new separation with a better weight is found, it can be possible that one of the known separations is dominated by the newly found separation. If that is the case, the dominated separation can be deleted. This leads to the optimization algorithm 6

Algorithm 5: pushBlueEdge

```
Data: Separation (X, Y)
Result: Separation (X_new, Y_new)
bestWeight ⟵ weight(X, Y);
bestSeparation ⟵ (X, Y);
for x ∈ parent(X) \ parent(X, Y) do
    for y ∈ parent(Y) \ parent(X, Y) do
        if weight(x, y) < bestWeight then
            continue;
        if isWellSeparated(x, y) then
            bestWeight ⟵ weight(x, y);
            bestSeparation ⟵ (x, y);
return bestSeparation
```

Algorithm 6: optimizeSeparations

Data: Set of Separations S
Result: Optimized Separations S_{new}
for $(A, B) \in S$ **do**
  $(A_{new}, B_{new}) \longleftarrow pushBlueEdge((A, B))$
  $S \longleftarrow S \setminus (x, y) \mid (x, y) \in S \wedge (x, y) \preceq (A_{new}, B_{new})$
return S

2.4 Distance Query

When given two nodes $u, v \in V$ it should be possible to get the shortest path distance as fast as possible. To do this, for every node u all the separations (A, B) are stored where $u \in A \vee u \in B$. Using this, it is trivial to access all the separations of a node which can be used to answer distance queries. In order to do this the separation for which holds $u \in A \wedge v \in B$ or $v \in A \wedge u \in B$ needs to be found.

2.4.1 Linear Scan

When given all separations of node u, the separation which can answer the query u, v can be found by searching through all the given separations.

2.4.2 Z Order Curve

not implemented
When the gridgraph has a hight and a width of 2^n for any $n \in \mathbb{N}$ a Z-Curve can be used to map from a 2D point v to a number in $\mathbb{N}$ such that the prefixes of the number in binary format can be used to identify the quadtree nodes which the node v is a member of.

2.4.3 Quadtree Prefix

not implemented
Since not all all the gridgraphs have a width and a hight of 2^n for any $n \in \mathbb{N}$, the quadtree can be used to assign numbers from $n \in \mathbb{N}$ for every node $v \in V$, such that the prefixes of the numbers identify the quadtree nodes a gridgraph node is a member of. These prefixes can be used to build a descision tree for each set of separations such that every separation for any node pair u, v can be found in $\mathcal{O}(\log n)$. Since only 64-bit numbers are being used, the decision tree can only be that high and the search can be considered $\mathcal{O}(1)$

3 Node Patches

Node selections are set of nodes (A, B) such that:

$$\exists x \in V : \forall u \in A : \forall v \in B : x \in \pi(u, v)$$

To find such sets, the path between two random nodes (a_0, b_0) is considered and then a node in the middle of the path is used as the barrier node. The sets $\{a_0\}$ and $\{b_0\}$ can then be used to grow sets of nodes. In an alternating way neigbours of nodes in set A and B are looked at and then checked if they fullfill the property that all shortest paths from the node to the other set of nodes can go over the barrier node x. If that is the case the node is added to the set. The random start nodes can be choosen from a set containing all $\binom{V}{2}$ nodes, and node patches should be searched until for all node pairs $u, v \in V$ there exists a node patch which can answer the distance query of nodes u, v. Figure 3 shows such a node patch of two sets.

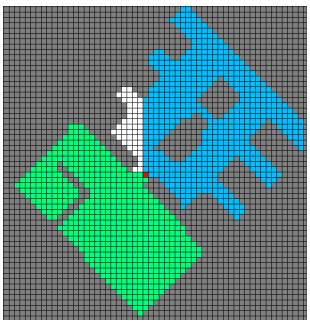

Figure 3: node patch example

3.1 Query

Each node patch stores its nodes (A, B) and the barrier node x. For each node n all node patches (A, B) where $n \in A \vee n \in B$ are stored sorted. Finding a node patch which can be used to answer a query $u, v \in V$ can be done in like it is done for HubLabels. When the node patch (A, B) with $u \in A \wedge v \in B \vee v \in A \wedge u \in B$ is found the query routine can be called recursivly for nodes u, x and x, v where x is the barrier node of the node patch (A, B). This query routine can be seen in Algorithm 7.

Algorithm 7: queryNodePatch

Data: Node Patche (A, B, x), Nodes u, v
Result: $d_{opt}(u, v)$
if $u \in neig(v)$ **then**
  return *1*
return $queryNodePatch(u, x) + queryNodePatch(x, v)$

3.2 Node Patch Optimization

not implemented

Since nodes have a common subsets of associated node patches (A, B) subsets of all node patches can be stored in buckets which then are stored by the nodes, instead of lists of node patches. In order to find optimal buckets $B \subseteq \mathcal{P}(S)$ of node patches for a set of separations S, a set of nodes with associated separations $N = \{(n, S_n) | n \in V, S_n \subseteq S\}$, the follwowing musst be assured:

$$\forall (n, S_n) \in N : \exists B_n \in B : \forall s \in S_n : \exists b \in B_n : s \in b$$

To get a best possible result for the distance query structure $|B|$ and $\forall (n, S_n) \in N : |B_n|$ musst be minimal, where B_n are the buckets node n stores.