525

Blocked All-Pairs Shortest Paths Dijkstra for Accelerating Grid-based Spatial Analysis

Using database-driven search of precomputed grid cell blocks

EKATERINA FUCHKINA, SVEN SCHNEIDER, MARTIN BIELIK

BAUHAUS-UNIVERSITÄT WEIMAR

ABSTRACT

Grids are a common representation for graph-based spatial analysis. For example, grid-based shortest paths are a good approximation for paths analysis and allow identification of central or segregated locations in a building. However, graphs built on grids can have a very large number of nodes and edges making computation costly and time-consuming. This, in turn, challenges the application of grid-based spatial analysis in the design process, where instant feedback on changes in the spatial configuration is desired.

This paper uses a Local Distance Database (LDDB) where distances between all possible positions of non-obstacle cells within a block are precalculated. The precomputed blocks allow to accelerating the Dijkstra algorithm with application to the All-Pairs Shortest Paths (APSP) problem. Thereby typical centrality measures can be calculated up to 4 times faster. The article will further investigate, how the block size affects the computation time, as well as the geometry of shortest paths and the subsequently computed graph centralities.

KEYWORDS

APSP, LDDB, Blocked Dijkstra, Betweenness centrality, Closeness Centrality

1 INTRODUCTION

Grids are a common representation for graph-based spatial analysis. For example, grid-based shortest paths are a good approximation for route finding (e.g., evacuation paths) or dynamic process simulations such as noise propagation or gas leak extension (Li et.al 2010). Moreover, graph centrality measures, which are derived on the shortest paths, such as closeness and betweenness are helpful in revealing strategic locations (e.g., remote places or highly frequented

places) which can be used to predict high-level user experience (Kim et.al 2009, Natapov et.al 2020).

However, graphs built on grids can have a very large number of nodes and edges. This makes it challenging to apply grid-based spatial analysis to large-scale spatial configurations. In the design process, where trial and error are highly demanded (Nisztuk et.al 2018), getting fast feedback on the effects of design changes requires faster calculations on the graph.
In our work, we will look at a grid partitioning method based on uniformly sized blocks (3x3, 4x4, or 5x5 cells), which allows us creating a so-called Local Distance Database (LDDB), where distances between all possible positions of non-obstacle cells within a block are precalculated. This approach results in a faster graph traversal: moving not between individual cells but between blocks of cells. While this method has been successfully applied for a Single Pair Shortest Path (SPSP) calculation (Yap et.al 2011) by adapting the A* algorithm, this paper will investigate its application potential for the All-Pairs Shortest Path (APSP) problem with the adaptation of Dijkstra algorithm. Acceleration of shortest paths calculation is followed by a faster closeness and betweenness centrality computation. Finally, we compare the resulting shortest paths with regard to classical APSP algorithm (Dijkstra, Floyd-Warshall) and discuss the effects on the centrality results.

2 LITERATURE REVIEW

For the analysis of spatial configurations, grids are a common representation. Thereby the architectural or urban space is overlaid by a regular grid. The grid cells are then assigned obstacle cells or non-obstacle cells, depending on whether they touch a physical object (e.g., wall or building) or not. The non-obstacle cells refer to the nodes in the graph, representing the points in space, that one can walk on. For defining the relationships between these nodes, two approaches exist – one based on visibility and another based on adjacency. In the visibility-based approach, all cells, that can be seen from a cell, are assigned as neighbors (Turner et.al 2001). In the adjacency-based approach, only the directly connected cells are assigned as neighbors (Kim & Choi, 2009). From the resulting graphs, different metrics can be computed, such as visibility-based closeness centrality, which highlights the easiest accessible spaces (least direction changes), or adjacency-based betweenness centrality, which highlights circulation paths in a configuration.

In this paper, we consider a grid-based graph based on adjacencies, where a cell can have maximum of 8 neighbors (so-called 8-neighbour graph or King's graph). Therefore, it is a sparse non-directed weighted graph. The best-known All-Pairs Shortest Path Algorithm for a sparse weighted graph is the Dijkstra algorithm run on each vertex (Karger et.al 1993).
A grid-based graph has many properties which allow accelerating the shortest path calculation. In the works of Zhang et al. (2016) and Li et al. (2020), rectangular spaces without obstacle cells are used to accelerate A* pathfinding. This form of space partitioning allows to "jumping over" such

spaces considering only border or corner cells of empty rectangles because inside of them the shortest path is known as a direct line. The graph pruning using the empty rectangles in a grid also can reduce symmetrical (equidistant but geometrically different) paths, which leads to more consistent shortest path results (Harabor&Botea, 2010).

In a work of Arge et al. (2001) grid partitioning on blocks is used to deal with large-scale grids, which do not fit into memory. Therefore, the grid is divided and stored in blocks, which sizes are adequate for an available RAM. This method allows to calculate the shortest paths first in each block locally and then running the Dijkstra algorithm to compute paths globally.

The approach suggested by Yap et al. (2011), also considers grid partition, but by uniform blocks of m×n cells size. Cells inside a block are precomputed in terms of all possible patterns of obstacle and non-obstacle cells. The shortest distances of each precomputed block are stored in a Local Distance Database (LDDB). The LDDB, therefore, is used to accelerate the A* single-pair shortest path algorithm. This approach will be discussed in detail in Section 3.1.

In a work of Sao et al. (2020) acceleration by partitioning is achieved by calculating the distance matrix divided on blocks, rather than subdividing the grid itself. This approach uses Min-Plus Matrix Multiplication and Cholesky factorization techniques on the Parallel Floyd-Warshall algorithm, which outperforms the Dijkstra algorithm on several classes of sparse graphs.

A grid is a discrete approximation of continuous space, which, as was mentioned above, can have many equidistant but geometrically unique paths in comparison to a single straight line between source and goal points (Nagy, 2020). Some spatial analysis approaches are sensitive to the geometry of the shortest path itself (e.g., betweenness centrality), other depend only on distance information (e.g., closeness centrality). Therefore, various optimization techniques applied to shortest paths algorithms on a grid can lead to different results in spatial analysis both in values and computation time. Consequently, it is an open area for research to investigate grid-based spatial analysis with regards to different pathfinding algorithms. In this work, we investigate block optimization for APSP Dijkstra algorithm and how the resulting shortest paths affect spatial analysis – closeness and betweenness centrality in particular.

3 LOCAL DISTANCE DATABASE (LDDB)

Local Distance Database (LDDB) is introduced in the work of Yap et al. (2011). The idea is that for each possible pattern of obstacle and non-obstacle cells (Figure 1a) within a block, the shortest path distance is stored between all boundary cells (Figure 1b). In our work, we will store a distance matrix between all cells inside a block. Additionally, a predecessor matrix is stored, which contains information about predecessor cells for shortest path reconstruction.

In general, all blocks can be divided into 3 types. First, isolated blocks, where all boundary cells are obstacle ones (Figure 1c). Second, disconnected blocks, where there is no shortest path between some non-obstacle cells within a block (Figure 1d). Third, connected blocks, where there exists a shortest path between all non-obstacle cells (Figure 1e). In our work shortest paths are built using octile (8-way) movements, though by the authors of LDDB (Yap et.al 2011) it is also considered 4-way movements or paths with realistic turn radius.

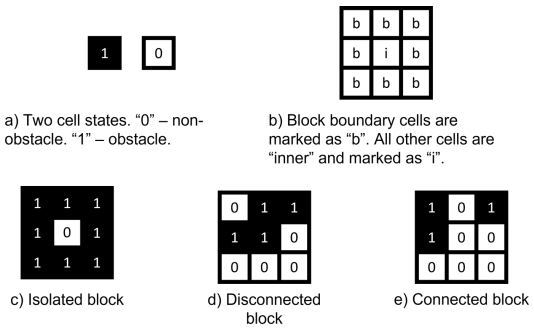


Figure 1: Block structure

In order to reduce database size, it can be reasonable to store matrices only for “unique” blocks. The other blocks then may be obtained by rotation and mirror actions from the unique ones. Since cells have only two states, each block is coded as a binary number where “0” represents a non-obstacle cell and “1” is an obstacle one. For example, to iterate through all possible combinations for a block of 3x3 size it is needed to start from “000000000” and end with “111111111” code, resulting in 512 possible combinations. A block can be added to a database only if other blocks obtained from it by mirror or rotation actions are not in the database already. Figure 2a illustrates an example, where block #30 will be added to a database and the other ones will reference this block to get distance and predecessor matrices transformed by a corresponding action (Figure 2b).

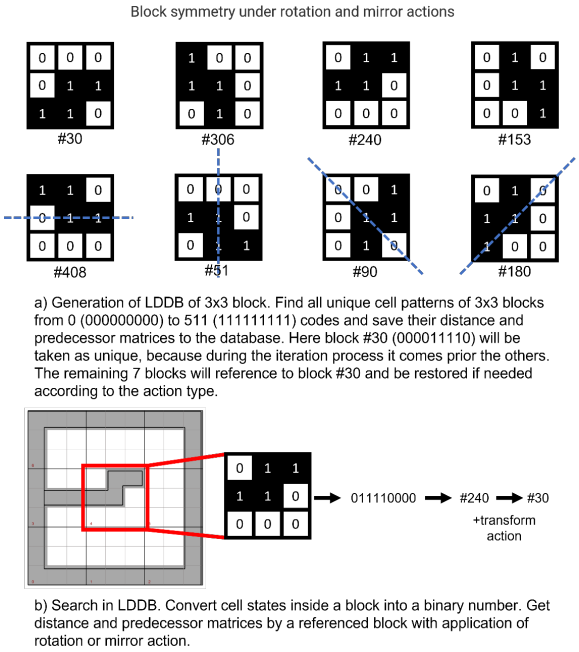


Figure 2: LDDB generation and search

Table 1 demonstrates the estimated size of the database depending on block dimension. The estimated number of unique blocks for various block sizes can be seen in "The On-Line Encyclopedia of Integer Sequences" [1]. A distance matrix is stored as a 4-byte array and a predecessor matrix as a 1-byte array. In our work Floyd-Warshall algorithm (Cormen et.al 2014) is used to compute the matrices. The distance matrix in turn can be also saved in a reduced form, where only half of the matrix is stored since distances are symmetric. Further, for isolated blocks, there is no need to store matrices, since for grid traversal such blocks will be ignored.

Block dim.	# All possible blocks	# "Unique" blocks	# Isolated blocks	Distance Matrix size	Reduced Distance Matrix size	Predecessor Matrix size
3x3	512	102	2	32,3 KiB	14,3 KiB	8,1 KiB
4x4	65 536	8 548	89	8,3 MiB	3,9 MiB	2,1 MiB
5x5	33 554 432	4 211 744	92 402	9,8 GiB	4,7 GiB	2,4 GiB

Table 1: Database size

3.1 Blocked A*: Single Pair Shortest Path

In the original work of Yap et al. (2011) LDDB is used to accelerate the A* algorithm for a grid tiled to blocks. A* is a single pair shortest path algorithm, that uses a heuristic function during the graph traversal process. First, consider a Breadth-first search (BFS) – a classical pathfinding algorithm for unweighted graphs. In BFS adjacent cells are queued for processing according to the depth level from the source cell: first, the neighbors of the source cell, then the neighbors of their neighbors, and so on. In contrast, in A*, cells will be sorted each time after enqueueing according to the priority, which is calculated based on a heuristic function. A simple example of such a heuristic function can be the Euclidean distance between the source cell and the goal cell (i.e., the length of line between these points). Using such a function, incoming neighbors are ordered according to proximity to the goal, which potentially leads to a faster pathfinding.

The same idea is used with the blocked A* approach, but the heuristic operations are made on the whole blocks rather than on individual cells. Therefore, before the algorithm starts, the grid must be subdivided into equal blocks for example of size 3 cells by 3 cells. Acceleration is achieved by the possibility to "skip" calculations inside the block (because they are precomputed) and to "move" inside the grid from boundary cells of one block to boundary cells of another one. Same as in A* blocks are inserted into the priority queue during an expansion process, where all valid neighbors of a block are enqueued according to a heuristic function. This priority of a block therefore can be calculated as a minimum Euclidean distance among all boundary cells to the goal cell. The process of expanding is repeated until the block containing the goal cell is reached and all remaining blocks in the queue have a priority larger than the found shortest path.

3.2 Blocked Dijkstra: All-Pairs Shortest Paths

In this paper, we use precomputed blocks to accelerate All-Pairs Shortest Paths (APSP) calculation by adaptation of the Dijkstra algorithm. APSP Dijkstra in general runs Single Source

Shortest Path (SSSP) Dijkstra as many times as there are cells (i.e., vertices) in a graph (graph must be without negative weights). Similarly, blocked APSP Dijkstra run blocked SSSP Dijkstra as many times as there are blocks in a grid. The expected result of the APSP algorithm is a global distance and predecessor matrices of V×V size, where V is the number of cells in a grid. Blocked Dijkstra uses the same idea as blocked A*, with the difference that there is no need for a goal cell heuristic because Dijkstra calculates distances from a source cell to all cells in the grid. Since the goal is all other cells, internal block cells are not "skipped" and their distances and predecessors are updated in the global matrices together with the boundary ones. Finally, the priority of a block is computed according to the minimum distance between its border cells and border cells of the current start block. Acceleration is achieved by checking the possibility to minimize distances between adjacent border cells of neighbor blocks. Only in the case if the distances between these boundary cells can be minimized – the checking process of minimizing distances between blocks' inner cells is run.

The distance between straightly connected cells is considered as 1 and between diagonally connected the distance is $\sqrt{(1^2+1^2)}=1.414$. This distance is referenced in our work as cost distance or CostDist. A distance stored in LDDB is a local distance between cells inside a block and denoted as LDDBDist. Finally, the calculated distance between cells of different blocks is stored in the global distance matrix and referenced as GlobDist.

Figure 3 demonstrates the process of calculating shortest path distances by blocked Dijkstra for block #0 in a grid that was divided into six blocks of 5x5 cells. In Figure 3a, cell IDs are depicted. The number in lower left of a block corner means a block ID. On the top of each step picture, there is a priority queue (Q) state, where blocks with their ID are placed and their priority is written in brackets. Although only a distance matrix is depicted and discussed in the example, the predecessor matrix is also filled during computation, saving the information about the cell from which the current path came from in the previous expansion step.

In Figure 3b, step 0 of the calculation is shown, where the source block is initialized into a global distance matrix and is enqueued in Q. By default, all distances globally are infinitely large. On this initialization step, distances from LDDB for block #0 are copied to the global distance matrix. Now, for example, the distance from cell #0 to cell #27 is set to "4.24". Here calculations are shown in relation to cell #0, but on each step, the algorithm calculates the distances also for all other cells in the block (#1, 2, 3, 8, 9, 10, 11, 16, 17, 18, 19, 24, 25, 26, 27) as well.

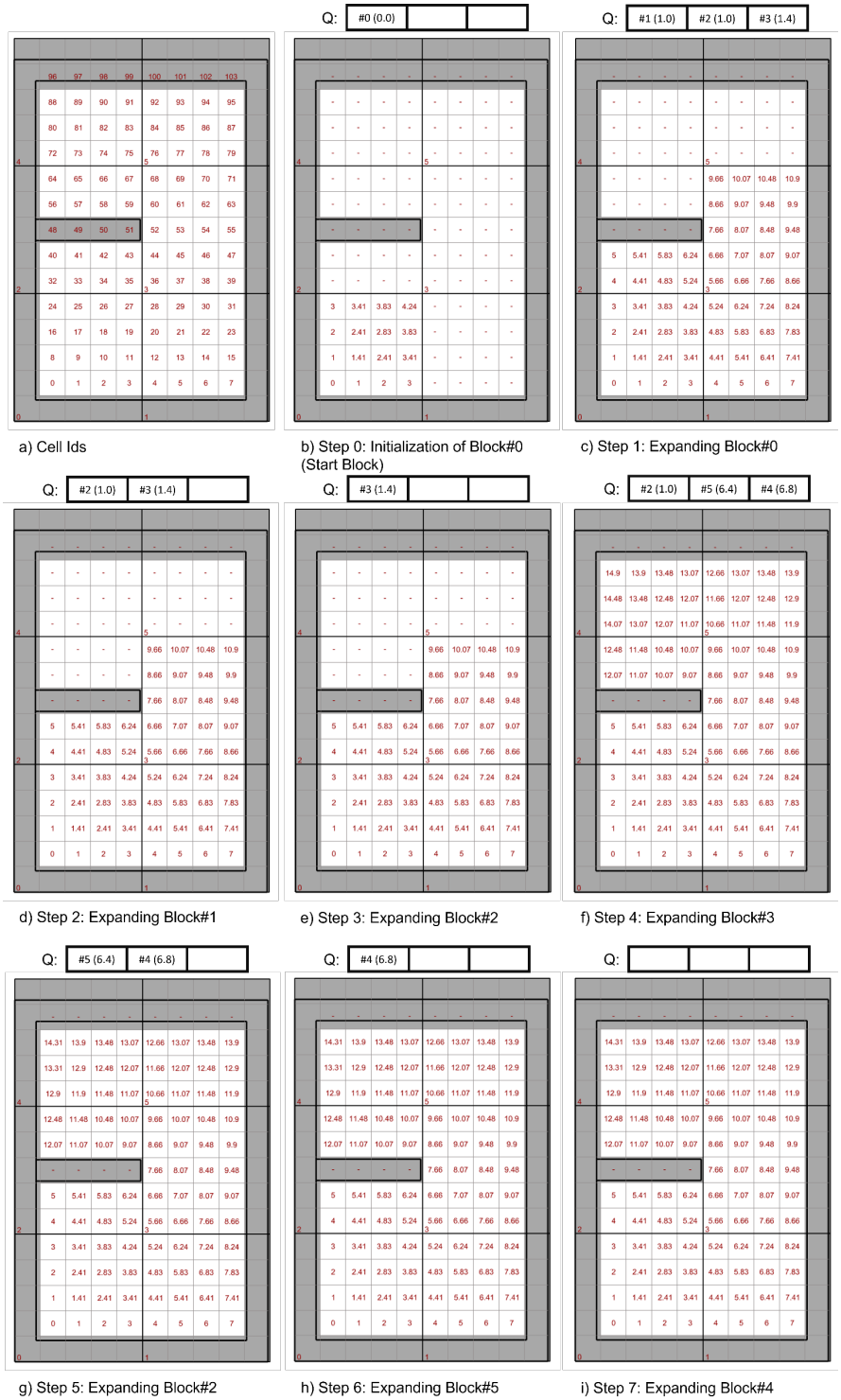


Figure 3: Blocked Dijkstra visualisation from block #0 and cell #0.

In step 1 (Figure 3c), the source block is dequeued and expanded. The expansion process includes checking valid neighbor blocks and propagating distances to their cells from all the cells inside the source block. For block #0 valid neighbor blocks are #2, #3 and #1. Between blocks #0 and #2 the next precomputed valid border cells adjacencies are loaded to check: #24→(#32,#33), #25→(#32,#33,#34), #26→(#33,#34,#35), #27→(#34,#35). Then the calculation of shortest path distances is started by trying to minimize the distance from cell #0 through cell #24 to cell #32.

The cost distance between cells #24 and #32 is “1”, the local distance between #0 and #24 is “3” and since before the global distance between #0 and #32 was set to default infinity and because $3+1=4 < \infty$ the algorithm will update the global distance between cells #0 and #32 to “4”. The fact, that a new shorter path is found through border cell #24, gives the algorithm a green light to check all the other cells inside block #2 from cell #0. For example, from cell #0 to cell #40 the distance will be calculated as: GlobDist(#0,#24) + CostDist(#24, #32) +LDDBDist(#32, #40) = 3+1+1 = 5. Notice, that not all the cells of block #2 are accessible (e.g. #56, #57…). They will be calculated when a path through some other block will be found. This process is repeated for all valid border cell adjacencies. The result of expansion and calculated distances for cell #0 is in Figure 3c. The neighbor blocks are enqueued, with corresponding priority. Both blocks #1 and #2 have the same priority “1” because the minimum distance between all border cells of source block #0 and these neighbor blocks is “1”. In turn, block #3 has priority “1.4”, because the only border distance between block #0 and block #3 lies between cells #27 and #36 with a diagonal cost distance between them “1.414”.

Step 2 in Figure 3d shows the expansion process of block #1. For cell #0 it cannot minimize any distances in neighbor blocks #2 and #3 and, therefore, no blocks are added to the queue. For example, when the algorithm will check the adjacency between cells #28 and #36, which is GlobDist(#0,#28)+CostDist(#28, #36)=5.24+1=6.24, it will turn out that the already set global distance between #0 and #36 is “5.66” through the cell #27 and since $5.66 < 6.24$, the path through the cell #28 to #36 improves nothing. The same happens in step 3 (Figure 3e) when block #2 is expanded: none of its neighbor blocks #3 and #1 can be improved.

Figure 3f shows step 4 with the expansion of block 3. It finally fills non-accessible cells from block #2 together with the other neighbor blocks #4 and #5. All these blocks are added to the queue according to the minimum distance between their border cells. For example, block #5 is added with priority “6.4”, because this is the distance between closest border cells #27 and #76. In the step 5 (Figure 3g), block #2 is expanded a second time. This will improve distances from cell #0 to cells #72, #72, #74, #80, #81 and #88. Since block #4 is already in queue and the priority also not changed –no actions are needed. Distances from block #2 to block #5 can’t be minimized and the algorithm goes to the final two steps, where blocks #5 and block #4 are expanded, and since no better paths (minimum distances) will be found, a queue will appear to be empty and algorithm finishes work here. The resulting distances on the image are the shortest path distances from #0 to all the other cells in the grid. This process is repeated for all cells of block #0 and afterwards for each block.

4 RESULTS

The blocked APSP Dijkstra is tested with block sizes 3x3, 4x4, and 5x5 using two different test cases (see Figure 4). Test Case 1 is the ground floor of a home for elderly people (Ishigami et.al 2012). Test Case 2 is a standard floor of an office building. Both cases differed in geometrical

complexity and grid size (4884 cells in Test Case 1 and 10787 cells in Test Case 2). The performance of the blocked APSP Dijkstra is compared to a classical APSP Dijkstra ("no" blocks) algorithm with a priority queue.

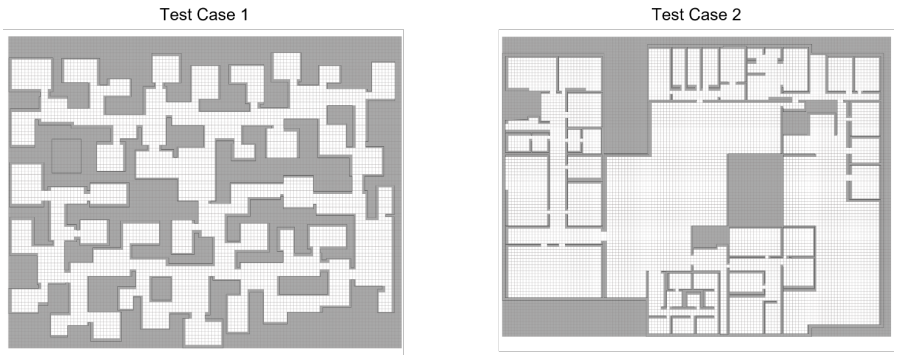


Figure 4: Test cases

As can be seen from Figure 5, any block version outperforms the classical algorithm on average by 4,5 times if considering the calculation of only shortest paths (SP) or shortest paths with closeness centrality (CC). If betweenness centrality (BC) is included in the calculation, the time of the blocked version is increased on average by about 3 times, while the classical version has only slight growth in time. It happens, because our current implementation BC is using the Brandes algorithm (Brandes, 2001), which takes advantage of the order in which cells are processed during the Dijkstra algorithm. Since in the blocked version only the order of blocks is tracked, and some blocks can be processed several times (see Figure 3g), there is a need for additional time for sorting cells by calculated distance in descending order before the BC algorithm can be run (see Appendix).

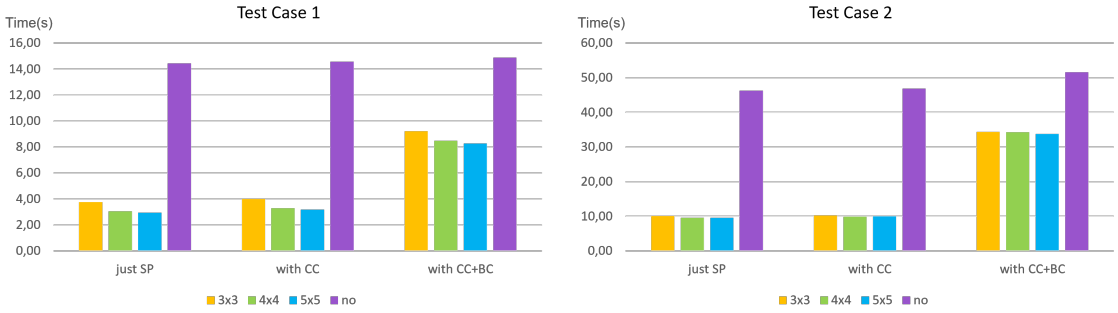


Figure 5: Comparing the performance of blocked Dikstra algorithm with the no-block version

		Test Case 1				Test Case 2			
Block Dimension		3	4	5	no	3	4	5	no
# Cells		4 884				10 787			
Time (s)	just SP	3,75	3,06	2,96	14,45	10,07	9,62	9,60	46,26
	+ CC	4,00	3,30	3,18	14,59	10,27	9,90	9,91	46,88
	+ CC and BC	9,23	8,48	8,28	14,90	34,39	34,30	33,80	51,56
# Blocks		1240	690	456	-	1419	800	520	-
# Disconnected Blocks		1	7	5	-	6	60	113	-

Table 2: Comparison of classical APSP Dijkstra with its blocked version including closeness (CC) and betweenness (BC) centralities calculation.

The calculation time of the blocked version is also depending on the number of disconnected blocks in a grid because such blocks needed to be processed several times and they cause neighbour recalculation chains. This is demonstrated in Test Case 2, where there is no significant acceleration for 5x5 block compared with 4x4 because the proportion of disconnected blocks is higher in 5x5 than in 4x4 (520 to 133 in comparison to 800 to 60 as shown in Table 2). The higher the block dimension, the higher the probability to discover disconnected blocks in a grid.

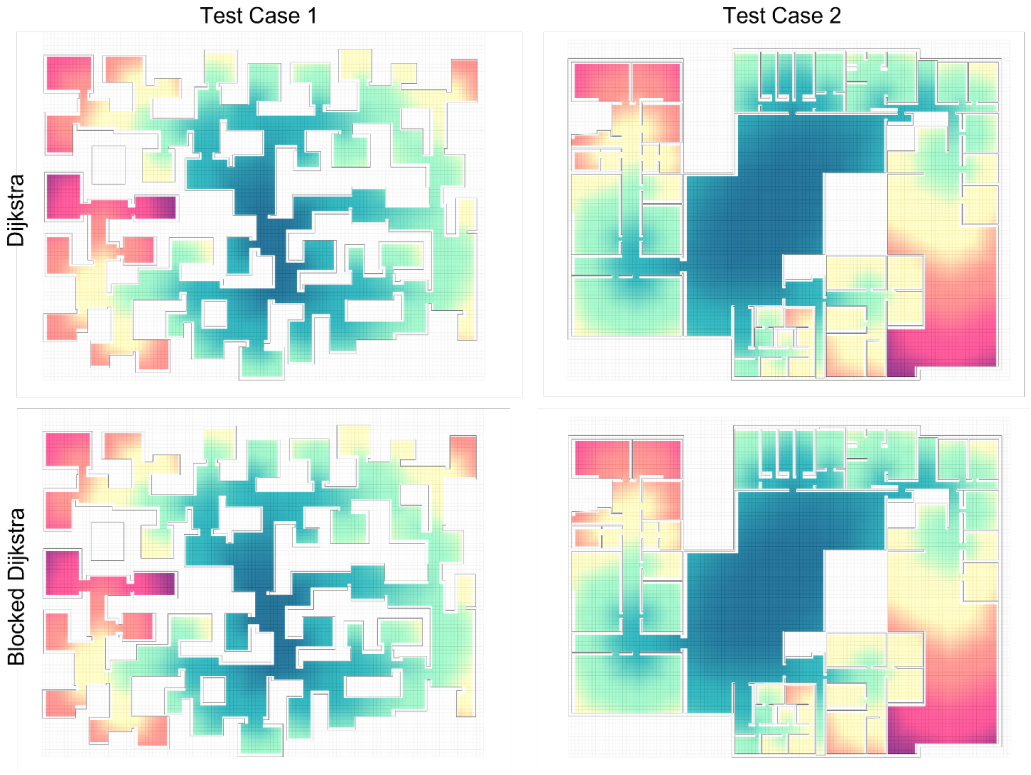


Figure 6: Closeness Centrality

Block Dijkstra algorithm and the classical one – both calculate identical shortest paths in terms of distances. In Figure 6 visualizations of closeness centrality for the two test cases are depicted. Since this centrality accumulates distances of all shortest paths available for a particular cell to others, there is no difference in the result of the calculation.

This is different from betweenness centrality, which calculates how frequently a particular cell participated in the shortest paths. First, forward and backward paths can be different, depending on the algorithm (see Figure 7b, 7c), moreover, different algorithms will return geometrically different paths with the same distance in a grid as shown in Figure 7. This is possible because on a grid there are many ways on how shortest paths can be planned from one cell to another. For example, the whole range of cells which can be used for the shortest path between source and goal cells are drawn in Figure 7e. The question is therefore which of the cells should be selected for the shortest path. Ideally, they should be equal to the lines resulting from the shortest path algorithm based on Any-Angle adjacencies, where these lines are “pixelized” by the Bresenham’s algorithm (Bresenham, 1977) shown in Figure 7a.

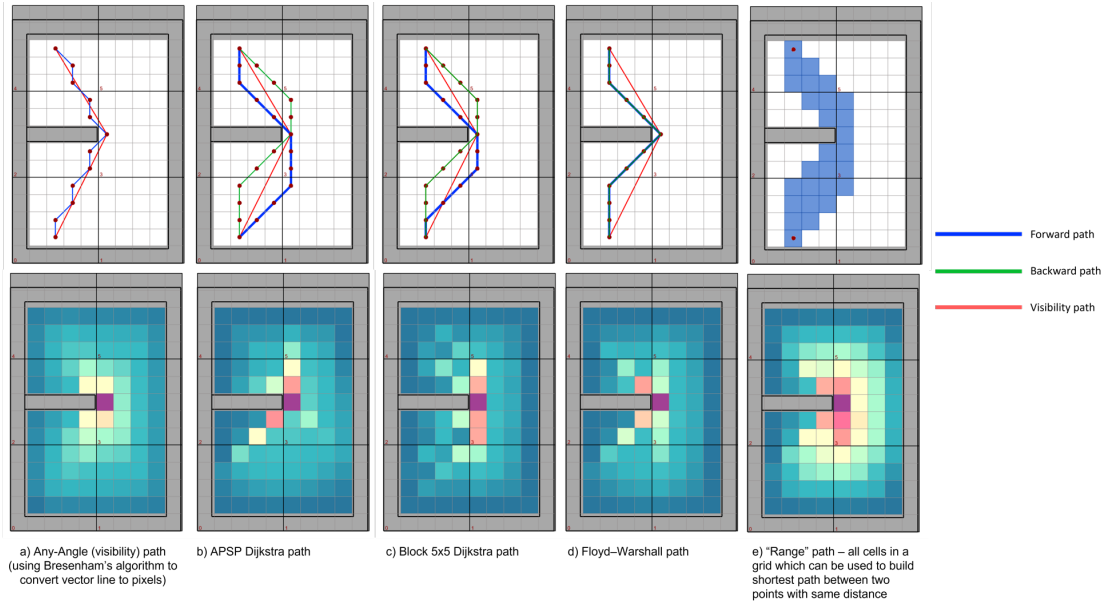


Figure 7: Betweenness Centrality in a grid depending on the shortest path algorithm

Even though for betweenness calculation only forward paths can be used because they are meant to be the same as the backward ones, it will not solve the problem, since it is not guaranteed by all the algorithms, that sub-paths which potentially should be equal will also be so, when the source and goal cells are not the same. This can be seen in Figure 8 for Test Case 2, where the central empty space has many diagonal tracks instead of one.

In Figure 8, all variants of betweenness centrality with regards to selected algorithms are shown. For a better comparison betweenness centrality obtained by pixelization of any-angle paths and by overlapping shortest paths ranges (see Figure 7 a,e) are also presented, though the calculation details are out of the scope of this paper. Despite the differences in the constructed shortest paths in the presented algorithms, they nonetheless approximate the critical areas on the plans quite acceptable. Shortest paths that are built by blocks tend to be more biased, and for the 3x3 block the spread seems to be higher in comparison to the 5x5 block.

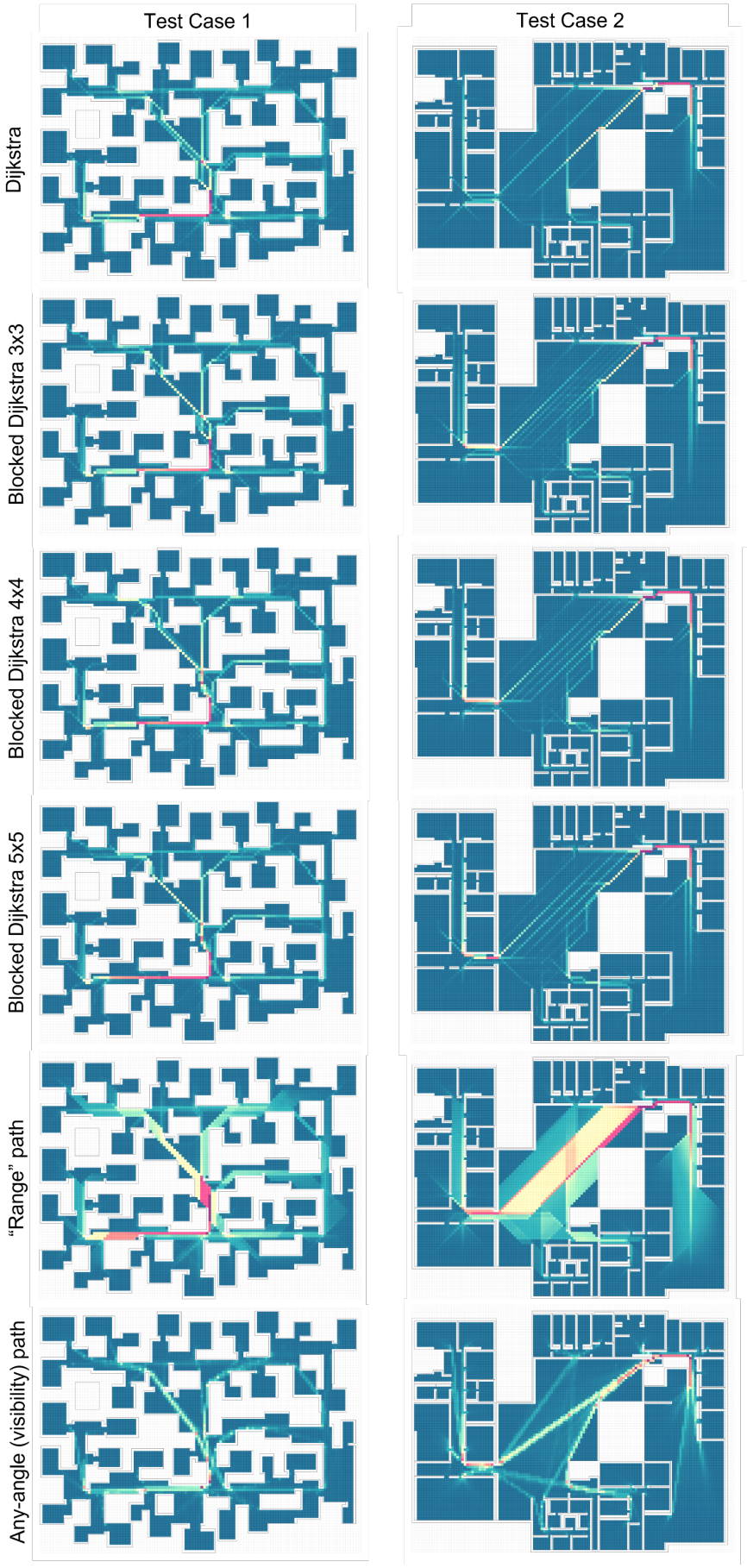


Figure 8: Betweenness Centrality

5 CONCLUSIONS

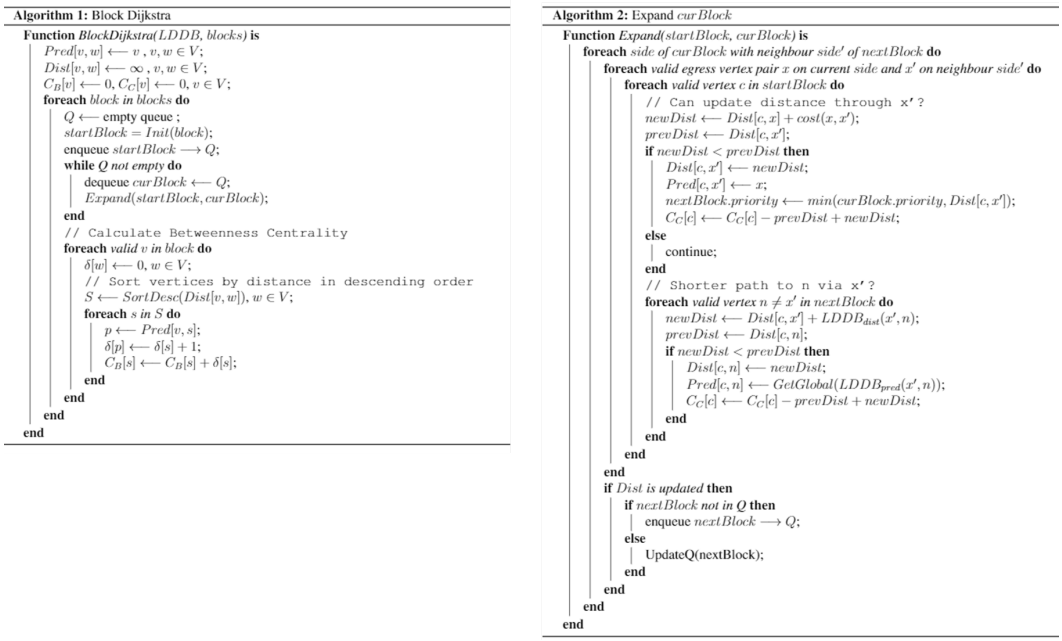
Grid-based spatial analysis is a useful instrument in the design process. Presented closeness and betweenness centralities calculations in this paper are accelerated using the Block APSP Dijkstra algorithm, which on average 4.5 times faster than the classical algorithm if considering only shortest path calculation and in general 1.5 faster with closeness and betweenness centralities included. Even though the larger block size can decrease the calculation time, in our test cases the time difference between 3x3 and 5x5 blocks is not significant. Especially considering how huge the difference in required memory is between the block size variants, it can be concluded that the 3x3 block size is the most reasonable choice. Considering the quality of centralities analysis, while the blocked closeness results have no difference in comparison to a classical Dijkstra algorithm, the betweenness centrality differs noticeably. The resulted betweenness based on blocked shortest path tends to be more inconsistent and spread compared to the classical Dijkstra algorithm. Nevertheless, the approximation of the critical areas on the plans remains informative.

Speaking of further possible improvements, there are versions of the Dijkstra algorithm which are run parallel and GPU accelerated (Crauser et.al 1998). Such methodologies can be potentially applicable to the blocked version as well. For further acceleration, some grid properties can be used, like path symmetries or empty rectangular spaces, for which calculations can be simplified (Zhang et.al 2016).

Additionally, the potential of the block algorithm is of high interest in the context of dynamic graph algorithms, which are used to avoid redundant calculations. This class of algorithms is aimed to update only those parts of the graph, which were affected by changes. Precomputed blocks then can locally update the distances and further update only paths lying through the changing area. This would allow geting results of spatial analysis faster within a constantly changing design.

6 APPENDIX

Pseudo code of blocked APSP Dijkstra algorithm with closeness and betweenness centralities.



Algorithm 1: Block Dijkstra

```
Function BlockDijkstra(LDDB, blocks) is
    Pred[v, w] ⟵ v , v, w ∈ V;
    Dist[v, w] ⟵ ∞ , v, w ∈ V;
    C_B[v] ⟵ 0, C_C[v] ⟵ 0, v ∈ V;
    foreach block in blocks do
        Q ⟵ empty queue ;
        startBlock = Init(block);
        enqueue startBlock ⟶ Q;
        while Q not empty do
            dequeue curBlock ⟵ Q;
            Expand(startBlock, curBlock);
        end
        // Calculate Betweenness Centrality
        foreach valid v in block do
            δ[w] ⟵ 0, w ∈ V;
            // Sort vertices by distance in descending order
            S ⟵ SortDesc(Dist[v, w]), w ∈ V;
            foreach s in S do
                p ⟵ Pred[v, s];
                δ[p] ⟵ δ[s] + 1;
                C_B[s] ⟵ C_B[s] + δ[s];
            end
        end
    end
end
```

Algorithm 2: Expand *curBlock*

```
Function Expand(startBlock, curBlock) is
    foreach side of curBlock with neighbour side' of nextBlock do
        foreach valid egress vertex pair x on current side and x' on neighbour side' do
            foreach valid vertex c in startBlock do
                // Can update distance through x'?
                newDist ⟵ Dist[c, x] + cost(x, x');
                prevDist ⟵ Dist[c, x'];
                if newDist < prevDist then
                    Dist[c, x'] ⟵ newDist;
                    Pred[c, x'] ⟵ x;
                    nextBlock.priority ⟵ min(curBlock.priority, Dist[c, x']);
                    C_C[c] ⟵ C_C[c] − prevDist + newDist;
                else
                    continue;
                end
                // Shorter path to n via x'?
                foreach valid vertex n ≠ x' in nextBlock do
                    newDist ⟵ Dist[c, x'] + LDDB_dist(x', n);
                    prevDist ⟵ Dist[c, n];
                    if newDist < prevDist then
                        Dist[c, n] ⟵ newDist;
                        Pred[c, n] ⟵ GetGlobal(LDDB_pred(x', n));
                        C_C[c] ⟵ C_C[c] − prevDist + newDist;
                    end
                end
            end
        end
        if Dist is updated then
            if nextBlock not in Q then
                enqueue nextBlock ⟶ Q;
            else
                UpdateQ(nextBlock);
            end
        end
    end
end
```

Figure 9: Blocked APSP Dijkstra Algorithm with CC and BC calculation

REFERENCES

Arge, L., Toma, L., & Vitter, J. S. (2001). 'I/O-efficient algorithms for problems on grid-based terrains', *Journal of Experimental Algorithmics (JEA), 6, 1-es*.

Brandes, U. (2001) 'A faster algorithm for betweenness centrality', *Journal of mathematical sociology*, *25*(2), pp. 163-177.

Bresenham, J. (1977) 'A linear algorithm for incremental digital display of circular arcs', *Communications of the ACM*, *20*(2), pp. 100-106.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009) 'Introduction to algorithms', *MIT press, pp. 97-117*.

Crauser, A., Mehlhorn, K., Meyer, U., & Sanders, P. (1998, August) 'A parallelization of Dijkstra's shortest path algorithm', In *International Symposium on Mathematical Foundations of Computer Science,* pp. 722-731.

D. Karger, D. Koller, and S. Phillips. (1993) 'Finding the hidden path: time bounds for all-pairs shortest paths', *Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pp. 560–568.

Hanauer, K., Henzinger, M. and Schulz, C. (2021) 'Recent advances in fully dynamic graph algorithms', *arXiv preprint arXiv:2102.11169*.

Harabor, D., & Botea, A. (2010, October) 'Breaking path symmetries on 4-connected grid maps', In *Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Ishigami, J. (2012) 'Group House'. *Seniorenwohnheim. ARCH+, 208*, pp. 128-133.

Kim, M., & Choi, J. (2009) 'Angular VGA and cellular VGA: An exploratory study for spatial analysis methodology based in human movement behavior', In *Proceedings of the 7th International Space Syntax Symposium,* pp. 054.1-054.14.

Li, C., Jian, L., & XueQuan, L. (2020). 'Static rectangle expansion A* algorithm for pathfinding', *IEEE Transactions on Games*.

Li, X., Claramunt, C., & Ray, C. (2010) 'A grid graph-based model for the analysis of 2D indoor spaces*', Computers, Environment and Urban Systems, 34(6),* pp. 532–540.

Nagy, B. (2020) 'On the number of shortest paths by neighborhood sequences on the square grid', *Miskolc Mathematical Notes*, *21*(1), pp. 287-301.

Natapov, A., Kuliga, S., Dalton, R. C., & Hölscher, C. (2020) 'Linking building-circulation typology and wayfinding: design, spatial analysis, and anticipated wayfinding difficulty of circulation types', *Architectural Science Review*, 63(1), pp.34-46.

Nisztuk, M., & Myszkowski, P. B. (2018). 'Usability of contemporary tools for the computational design of architectural objects: Review, features evaluation and reflection', *International Journal of Architectural Computing, 16(1)*, pp.58-84.

Sao, P., Kannan, R., Gera, P., & Vuduc, R. (2020, February) 'A supernodal all-pairs shortest path algorithm', In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming,* pp. 250-261.

Turner, A., Doxa, M., O'Sullivan, D. & Penn, A. (2001) 'From isovists to visibility graphs: A methodology for the analysis of architectural space', *Environment and Planning B: Planning and Design, 28(1)*, pp.103-121.

Yap, P., Burch, N., Holte, R. C., & Schaeffer, J. (2011, August) 'Block A*: Database-driven search with applications in any-angle path-planning', *In Twenty-Fifth AAAI Conference on Artificial Intelligence*.

Zhang, An, Chong Li, and Wenhao Bi. (2016) 'Rectangle expansion A∗ pathfinding for grid maps', *Chinese Journal of Aeronautics* 29.5, pp. 1385-1396.

[1] http://oeis.org/search?q=1%2C2%2C6%2C102&language=english&go=Search