Task Planning for Autonomous Drilling Excavators Under Terrain Constraints

Marcel Suiker[1], Jörg Husemann[1] and Karsten Berns[1]

[1]Robotics Research Lab, Department of Computer Science, RPTU University Kaiserslautern-Landau, Germany

marcel.suiker@cs.rptu.de, joerg.husemann@cs.rptu.de, karsten.berns@cs.rptu.de

Abstract -

Task planning is essential for the operation of autonomous machines. For autonomous construction vehicles, such as drilling excavators, these tasks not only need to be performed in an efficient order but also must take the terrain into account. This work presents an approach to analyzing terrain information based on environmental topology and obstacles, leveraging systems such as clustering, Sobel Filters, and candidate-area maps. The analysis is further used to plan drilling operations, additionally taking the robot's kinematics and desired drilling points into account. The concept is tested and evaluated in a simulation using Unreal Engine 5. Experiments showed that the presented approach can plan an efficient task order based on terrain, enabling productive execution with autonomous drilling excavators.

Keywords -

Task Planning, Robotics in Construction, Terrain Analysis

1 Introduction

Due to the lack of human resources in the construction industry [1], a need emerged to automate previously human-operated construction machines. Efficient automation leads to more precise work, which can guarantee the same high quality in each iteration of the robot's tasks. For efficient automation, not only the machine itself, but also the associated tasks, need to be automated. This includes task planning, which was previously performed by a human and judged based on their knowledge. It is of utmost importance to automate the task planning itself, as autonomous intelligent machines are in need of changing their task plans during the operation based on the information of the environment. In some cases, it is even beneficial to replan the tasks entirely. Current task planning algorithms do not account for vehicles with high degrees of freedom that are in need of precise movements. Furthermore, the solutions do not provide task plans that group tasks together, allowing grouped tasks to be performed from a single location. This paper aims to provide a framework for robot operation planning. It considers the terrain constraints, the robot's kinematics, the closeness of drill holes, and the travel distance between them. The framework groups tasks, provides feasible execution regions prior to task execution, and optimizes the task execution order to minimize travel paths. An autonomous drill excavator, Morath BR8010, figure 1, is used as the experimentation platform of this paper. The drill is a differential-driven vehicle with tracks capable of operating in challenging terrain. Its arm has seven degrees of freedom and uses hydraulics for movement. The robot is capable of moving heavy weights, even with the arm fully extended, but this comes at the cost of reduced operation speed and accuracy. Its main use cases are anchor drillings, renovation, and restoration of foundations and shoulder shorings. The paper is structured in the following way. Section 2 gives a brief overview of current task planning approaches in construction scenarios. After which, section 3 explains the algorithms used throughout the developed task planning approach. The approach to task planning itself is described in 4. Experiments in a simulation environment and the real robot are shown in section 5. Lastly, the paper is summarized and brought to a conclusion 6.

Figure 1. Morath BR8010

2 Related Work

Terrain-dependent task planning is an essential step in automating off-road vehicles like construction vehicles. Until now, most solutions focus on platforms operating in controlled, flat environments like warehouses [2, 3]. This led to them not performing any terrain analysis. Seo et al. propose a solution to task planning for automated excavators [4]. The approach takes into consideration

both the terrain and the requirements for effective excavation. It plans a path for the robot on how and where to dig. This is all done using layers and masses defined by the terrain's shape. Terenzi and Hutter present another approach to plan the operation of autonomous excavators [5]. It builds a three-dimensional representation of the construction using iterative closest point-based SLAM and registers it using GPS. For planning, the map is transformed to a grid map with layers for elevation, target elevation, occupancy, excavation, and dump areas. It calculates in which order to excavate the grid cells, where to dump the excavated soil, and the path and navigation of the excavator. Even though those approaches do take a terrain analysis into account, they are not applicable to drilling excavators. This is the case as both approaches present solutions for planning coverage tasks, while drills do not cover an area but precisely drill one hole at a time. Furthermore, neither approach accounts for the robot arm's kinematics, making it unable to verify whether the robot can reach a point from a base position. The spoon's orientation is also not taken into account when planning their tasks. This is necessary to ensure high-precision placement of the drill head to prevent the drill from jamming or drilling holes at the wrong angle.

3 Algorithms and Methods

Planning efficient tasks under terrain constraints is a complex problem. By splitting the problem into smaller sub-problems, planning becomes easier. This section focuses on presenting algorithms suitable for solving those sub-problems. First, approaches to clustering using Voronoi Diagrams 3.1 and computing the terrain's gradient 3.2 are presented. After which, the algorithms for computing the Candidate Area Map 3.3 and finding positions for the robot using the biggest inscribed rectangle 3.4 are proposed. The complexity classes of the algorithms are listed in table 1.

3.1 Distance-Based Clustering of Drill Holes

To increase the efficiency of the drilling operations, the drill holes reachable from a single position are grouped. To cluster the points efficiently based on their distance, Voronoi Diagrams can be used. Jumping Flood [6] is used to approximate the Voronoi Diagrams quickly with complexity of $\mathcal{O}(n \log n)$. A grid map where each cell represents a tuple $< s, position(s) >$ with s being the closest seed and its position is used by the algorithm. When initializing the algorithm, the drill holes are marked and assigned a unique seed and position. The algorithm constructs a Voronoi diagram in $\log n$ rounds, where n is the length of the grid map's sides, which can be seen in figure 2. The cells are updated if the distance from the incoming tuples' seed and position to the grid cell is smaller than the current assigned value. In each round, the step length k gets divided by two until it reaches $k = 1$. To join the cells by their distance, an algorithm by Koivistoinen et al.[7] is used. As the algorithm joins cells by volume, it is modified as can be seen in Algorithm 1 to support joining cells by distance. The algorithm efficiently computes clusters of arbitrary size and shape, which is necessary for planning the operation.

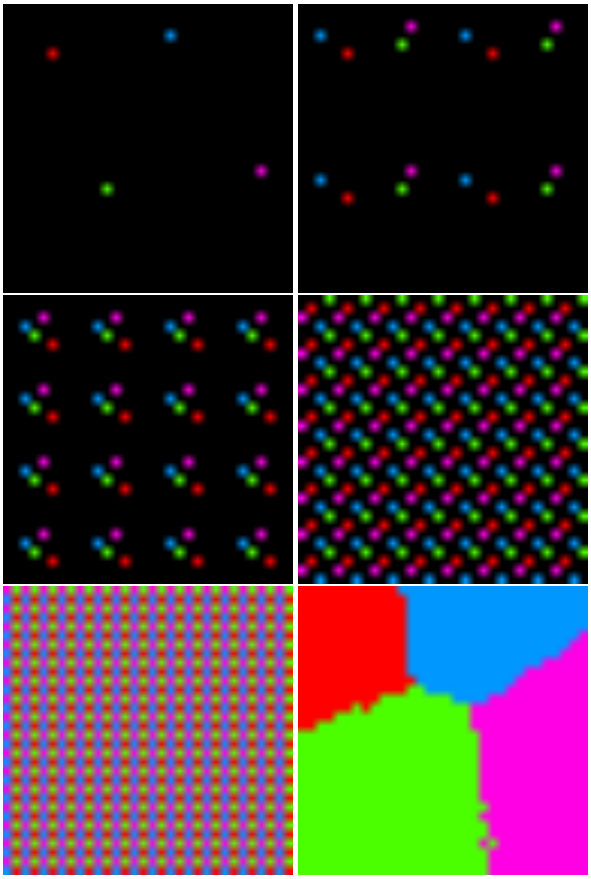


Figure 2. Creation of a Voronoi diagram for a grid map of size 32×32 cells. The figures show the steps that were taken to approximate the Voronoi diagram for four given points.

3.2 Computation of the Terrain's Gradient Matrix

To estimate the pose of the robot, it is essential to have knowledge of the slope of the terrain. The gradient of the terrain's height map represents the slope. Due to the expensive computation of gradients, they are estimated. To further reduce computational complexity, the terrain's height is represented by a reasonably sized grid map. Height maps can be represented as grayscale images; thus, the gradient can be computed efficiently and robustly using computer vision methods. Sobel Filters [8] are a proven strategy to estimate the gradient of

Algorithm 1: Modified Voronoi Clustering

```
Data: Set of points P, distance threshold d_max
Result: Set of cluster C
V = jumpingFlood(P);
Compute distances between cells R_i of V;
Order p_i by increasing distance of associated R_i
  ⇒ p_1, p_2, ..., p_{n-1}, p_n ;
label = 0;
for i = 1 to n do
    for j = 1 to i − 1 do
        if d_{C_i,R_j} <= d_max then
            if C_i is neighbor of R_j then
                Merge C_i and R_j;
                if p_j has no label then
                    Add label of C_i to p_j;
                else
                    Merge into cluster with
                      smaller label;
                end
            else
                label+ = 1;
                add label to p_j ;
                add p_j to cluster C_label;
            end
        end
    end
end
```

a pixel by considering its direct neighborhood. Sobel filters are represented by two 3×3 matrices

$$\begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{1}$$

1, which represent the weighted functions for x and y components of the gradient. In computer vision, they are used as convolutional kernels to find edges and corners. To compute the gradient matrix, first the kernel for the x-direction and the y-direction are applied consecutively to each pixel of the height map.

$$g_{ij} = \frac{g_{ij}}{(max(G) - min(G))} \quad g_{ij} \in G | i, j \in \mathbb{Z} \tag{2}$$

Next, the values are averaged and normalized to the range $[0, 1]$ with 0 representing plateaus and slopes having values higher than 0.

3.3 Computing the Candidate Area Map

The placement of the robot relative to a drill hole is restricted by the robot's size and workspace. To find a safe area for the robot's placement, the Candidate Area Map (*CAM*) is computed. For the computation of the *CAM*, the gradient map, an obstacle map, the positions of the m drill holes, and the robot's inverse reachability map (*IRM*) are taken into account. Obstacle maps are binary images in which black pixels represent areas occupied by obstacles. By considering each pixel to represent a grid cell, a grid map matching the image's resolution can be constructed. Cells represented by a black pixel are marked as occupied; all others are obstacle-free. The set of all positions of the robot from which it is able to drill a hole at a given pose (x, y, z, θ) is called *IRM*.By setting up a kinematic system where the end-effector is the root, and the undercarriage of the robot is moved. The *IRM* can be calculated iteratively by saving all possible reachable points in a voxel map. As multiple points of the *IRM* might lie inside a voxel cell, the number of points in the inverse reachability map is reduced. To ensure each voxel cell is correctly added to the *IRM*, they are classified by performing a neighborhood search with a search radius of $r = 2$. For the computation of the candidate area map, the obstacle map, the gradient matrix, and the inverse reachability map are joined.

Table 1. Overview of the complexity classes of the used algorithms.

Algorithm	Complexity	Reference
Clustering	$\mathcal{O}(n^2 + n)$	
Gradient Matrix	$\mathcal{O}(n^2)$	
CAM Computation	$\mathcal{O}(n^3 m)$	
Biggest Inscribed Rectangle	$\mathcal{O}(n^5 k)$	[9]
MPSO	$\mathcal{O}(nm^4)$	[12]

$$z = \frac{H_{i_x, i_y}}{v} \tag{3}$$

First, for all (x, y) coordinate pairs, a z-coordinate is computed using equation 3 with $H_{i_x i_y}$ being the value of the height map at a point (x, y) and the voxel size v. Then, the *CAM* is constructed as a binary image, where white pixels are below the slope threshold, obstacle and drill hole-free, and their coordinate (x,y,z) is part of the *IRM*; otherwise, the pixel is black. By marking the cells surrounding the drill hole at a distance equal to the undercarriage's length as black pixels, it is ensured that the robot can't be placed on top of the drill hole. *CAM*s can be joined to group candidate area maps (*GCAM*) by calculating the intersections of all *CAM*s belonging to the same cluster of drill holes as used in section 3.1. If the *GCAM* is smaller than the robot, the cluster of *CAM*s is split, and two new *GCAM*s are computed. This is repeated until a *GCAM* of feasible size is computed. The *GCAM* represents all points from which each drill hole of the cluster can be drilled.

3.4 Finding Positions Inside the GCAM

The next step is to determine the robot's position. As the *GCAM* represents all points from which the drill holes are reachable, it is feasible to find an inscribed rectangle inside its contour that is bigger than the robot. An orientation invariant rectangle can be found by defining the polygon on a grid $GL = \{(xi, yj) : 0 \leq i \leq r, 0 \leq$

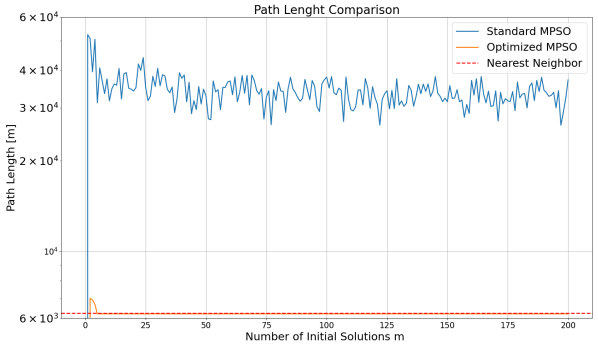
$j \leq s\}$ and lattice polygon P [9]. Furthermore, four sets are defined:

- the set of all points:
  $P = \{p_0, p_1, \ldots, p_N\} = \partial P - \bigcup_{i=1}^{h} (\imath H_i)$
- the set of all points of the polygon: $polygon = \partial P$
- the set of a single hole: ∂H_i
- the set of all holes:
  $holes = \partial H_0 \bigcup \partial H_1 \bigcup \partial H_2 \bigcup \cdots \bigcup H_i$

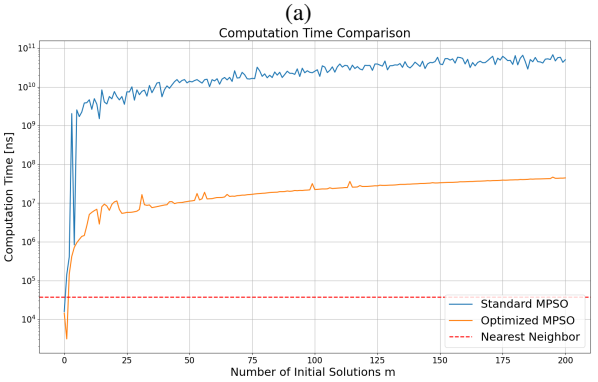
With ∂H_i being the contour of the hole i and $\imath H_i$ being the points inside the hole i [9]. First, the approach computes a symmetric adjacency matrix A, where $a_{i,j} = 1$ if there is an edge between i and j, else $a_{i,j} = 0$. The prerequisites to computing A are determining if a point p is inside ∂P or ∂H_i, and if the intersection between a segment and ∂P or ∂H_i is allowed or not. The adjacency matrix is computed by calculating the sides of the polygon and traversing all points of P, computing equation 4.

$$l = (P[I], P[j] \text{ with } i, j \in \{0, length(P)\}) \quad (4)$$

As A is a symmetrical matrix $A[i][j] = 1$ and $A[j][i] = 1$ if l is part of the border of the polygon. If l doesn't intersect with the polygon or a hole in a forbidden way, it is checked if the middle point m of l is inside the polygon. This results in $A[i][j] = A[j][i] = 1$. For polygons with holes, additional checks are performed. For each hole, iterate through its points and calculate the middle point between two points $H[i][j]$ and $H[i][k]$. If the middle point is not part of the boundary of $H[i]$, the values in the adjacency matrix are set to 0. The adjacency matrix is then used to compute the largest inscribed rectangle inside the polygon. The algorithm starts by computing a quadrilateral and checking whether it contains any holes, ensuring that the computed rectangle does not overlap any holes. Next function *SIDES* [9] computes the sides of the rectangle by calculating the segments connecting points p_1 and p_2 with distance $d = 3$. This is done by checking if $A[p_1][p_2] = 1$, meaning they form a valid edge and computing all edges p_1 and p_2 from distance $k = 1$ up to $k = d + 1$. Only lists of edges ending at p_2 will be used for further computations. Furthermore, the function rejects solutions if two edges intersect each other or if three consecutive points are collinear. Lastly, the function removes all lists of edges containing a hole. The largest rectangle is computed by iterating over all points in P such that two points $p_i = P[i]$ and $p_j = P[j]$ are picked with $i < j \wedge j \leq N$. Points p_i and p_j are used to compute the list of sides using *SIDES*. Non-empty lists are inserted into a list of all possible solutions. The solution forming the biggest rectangle is picked from the list, and all duplicates are removed. As the algorithm does not ensure that the quadrilaterals formed are rectangles, it is checked that three corners have an inscribed angle of $\frac{\pi}{2}$. As for rectangles, the fourth corner is indicated by the others. Using the calculated rectangle, the robot's pose can be computed as the rectangle's midpoint m_{rect}, and the orientation of its longest side θ_{rect}.



(a)



(b)

Figure 3. Comparison of the path length 3a and computation time 3b of the *MPSO*, *optimized MPSO* and *nearest neighbors heuristic* [10] for a dataset of 200 randomly scattered points.

3.5 Finding an Efficient Path for the Operation

An important aspect for the efficiency of the drilling is the paths between the drill spots. By traveling only once to the drill spots, the travel distance can be minimized. Finding a permutation that fits this criterion is known as the traveling salesman problem. Traveling salesman problem is defined as trying to find a permutation $P = 1 + \imath_1 + \imath_2 + \cdots + \imath_n$ of n integers with the lowest quantity $a_{1\imath_1} + a_{\imath_1\imath_2} + \cdots + a_{\imath_n 1}$ [11]. The $a_{\alpha\beta}$ indicates the cost function being optimized, e.g., travel time or distance. As large construction sites can consist of hundreds of drill holes, optimizing the travel path is necessary. Due to the traveling salesman problem belonging to the class of *NP*-hard problems, a heuristic solution is needed. A heuristic approach making use of a modified particle swarm optimization (MPSO) [12] is initialized by generating m random permutations of the nodes called solutions. Using an appropriate ob-

jective function, solutions are evaluated. The results of the function are used to determine the best value ever reached by a solution *pbest*, the best overall solution for the current computation round *gcbest*, and the best solution computed overall across all rounds *gbest*. If a random value is smaller than $\alpha \leq a_1 \leq \beta$ *pbest* moves linearly towards *gbest* if it is bigger towards *gcbest*. Value a_1 is inititaly equal to α and moves towards β. By finding sequences that occur in both *gcbest*, or *gbest*, and *pbest*, the solutions are either moved to *gcbest* or *gbest*. These sequences are then applied to the particle, and *gbest* and *pbest* are updated. If *gbest* or *pbest* changed, four local search algorithms are applied to the particle to find better solutions close to the current solution and avoid getting stuck at a nonoptimal solution. If a local search algorithm finds a better solution for the chosen metric, the current solution is replaced. If the termination criterion is met, the algorithm stops updating the solutions and terminates if *gbest* has not changed in ten consecutive iterations. The path of *gbest* is then returned as the solution to the TSM problem. Experiments 3 have shown that the presented solution does not yield reasonable results in a feasible time. The algorithm was optimized by exchanging the random initial solutions with permutations gathered using the nearest neighbor heuristic [10] with different start points. This small change results in a significant reduction in computation time and improves the solution, as shown in figure 3.

4 Terrain-Dependent Task Planning

To plan the tasks, the terrain's height map and obstacle map, the robot's *IRM*, and a list of coordinates for the planned drill holes are used 4. Using the approach presented in 3.1, drill holes are grouped such that they can be drilled from the same position without moving the robot. The groups are then stored in a list, which is used to compute the *GCAM* of each group as described in 3.2. As only the computation of the candidate area map needs to be done for each drill hole in the group, it is sufficient to compute the slope, read the obstacle map, and workspace information once. Using the *GCAM*s, the robot's poses are estimated 3.2. They are then stored in a list and associated with their groups. By adding the poses to a map administrator, an adjacency map is computed based on the number of grid cells between them. The distance between the poses is estimated by combining information about the slope and obstacles into a single grid map. Then, a wavefront algorithm is used [13] to calculate the shortest paths between two poses and their distance. It operates by sending a wave out from a start point. Each cell further away from this start point increases the distance by one, except obstacle cells, which have a distance of infinity. After reaching the goal position, the path is calculated starting at the

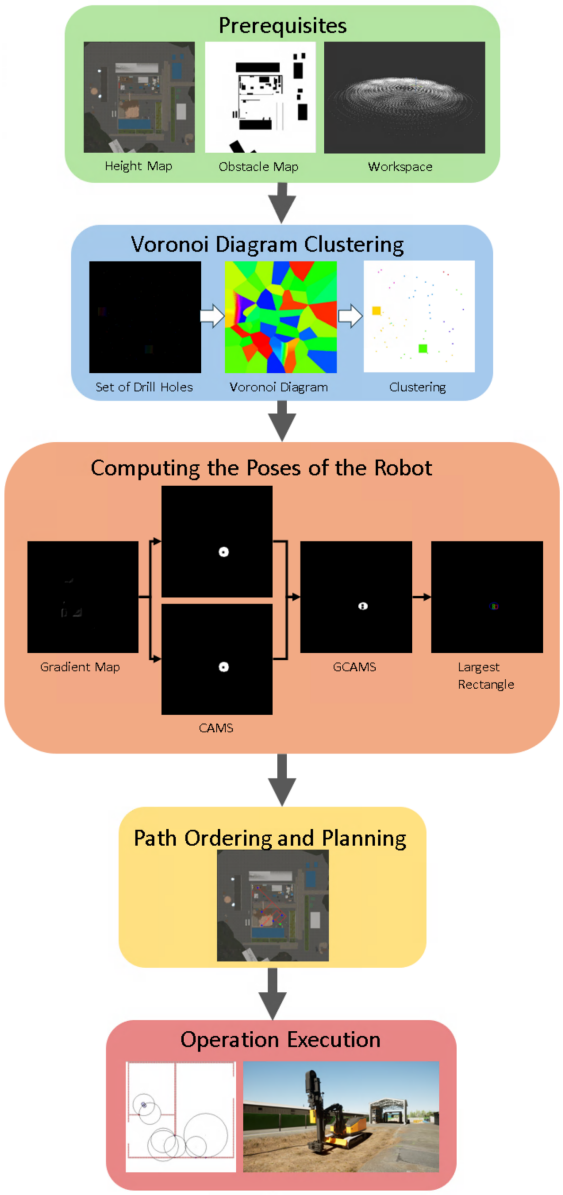
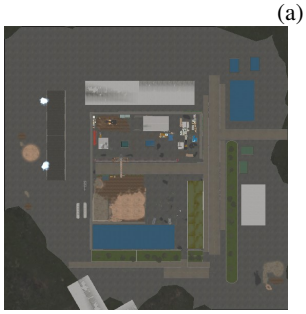
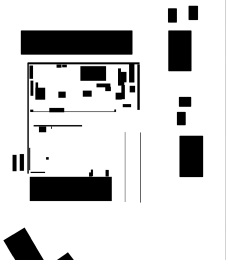


Figure 4. Overview of the Operation Planning.

goal, with the next cell always being the neighborhood cell with the lowest distance to the start. A path is determined when reaching the start cell. After calculating the path, its distance is written to the adjacency matrix. This adjacency matrix is then used as the adjacency matrix of the modified particle swarm operator 3.5. The MPSO algorithm then calculates the order in which the drill excavator travels to the estimated poses. To travel there, the elastic band's framework [14] is used, with the initial paths for the elastic band given by the paths calculated using the flood fill algorithm. The elastic band's framework can dynamically react to environmental changes and adjust its paths accordingly. This reduces the need for an online algorithm, since a change in the environment does not require replanning the paths and their

(a)





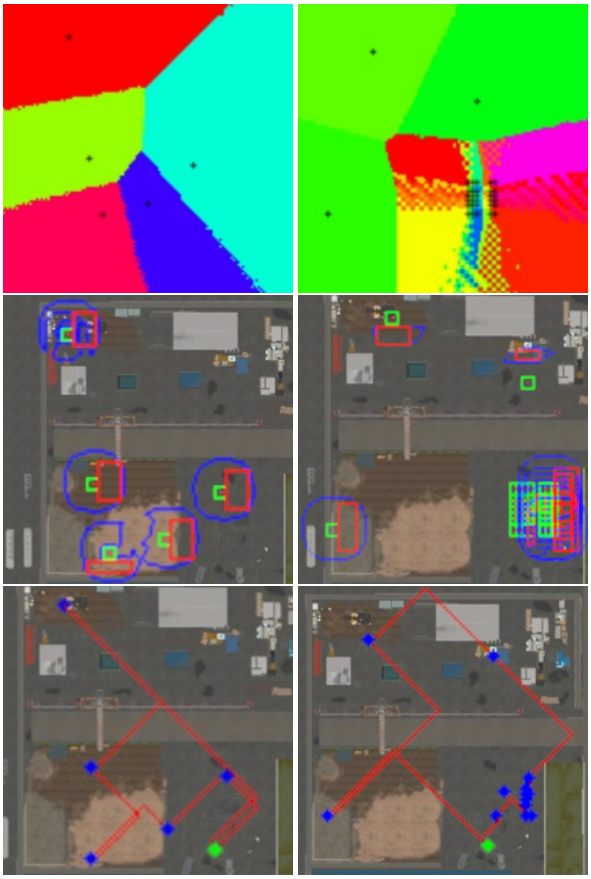
(b) (c)

Figure 5. Images of the used construction site, the first image 5a shows the construction site in Unreal Engine, the second one 5b is a top view of the construction site, and the third image 5c is the used obstacle map.

order. When the robot reaches an estimated pose, it can then perform the drilling operation. To move the arm to the desired position of the drill hole, the angles of the joints are computed by solving the arm's inverse kinematic problem for this position.

5 Experimental Results

The system was implemented using the FINROC framework [16] and simulated using Unreal Engine [17]. A map of a construction site with various terrains, including pits, steep slopes, and flat areas (Fig. 5), was used as the environment for the experiments. Multiple experiments were performed to evaluate the Operation Planning System. Before carrying out the experiments, the *IRM* of the robot was captured as a point cloud ; each of the points in the *IRM* corresponds to points that are reached by the robot while the drill is oriented orthogonally to the ground. Furthermore, the simulation environments' obstacle and height maps were calculated (Fig. 5). As a first experiment, a dataset consisting of the five points used in [15] was used (Fig.6a). When looking at the Voronoi diagram, each drill spot forms a distinct Voronoi Cell with slightly fuzzy edges. These distinct cells form five groups, as the distance between them exceeds the clustering distance of 6m. For the computed largest rectangles in the *GCAM*s, the algorithm finds reasonable solutions, all of which are large enough to place the robot inside. Almost all solutions are of the same or higher quality than the ones computed by Suiker et al. [15] and only need 15.93% of its computation time 2. Even more significant reductions in computation time can be seen in table 2, experiment 5, where the computation time was reduced to 4.96% of that of the version by Suiker et al. The reduction in computation time comes not only from more advanced, faster algorithms but also from lower resolutions for the *GCAM* contour. This creates the issue that can be seen in the uppermost drill hole; the resolution gets so small that the algorithm skips over some of the gaps in the concave shape of the *GCAM*. For the second experiment 6b, a set of 39 points was chosen, of which 35 were grouped at a distance of two meters with five columns and seven rows. The given set of points results in thirteen *GCAM*s, which are associated with one to six points. Due to the reduction in the resolution of the *GCAM* contour, its concavity is lost, leading to the computed pose being placed inside the contour's hole. In other cases, the al-



(a) (b)

Figure 6. The results of the first and second experiments. The first image shows the generated Voronoi diagram; the second shows the computed largest rectangles (red), the *GCAM* contour (blue), and the inaccessible area around the drill holes (green). The last shows the computed routing with the start and end point (green).

Table 2. Experiment Overview of nine experiments with eight different sets of drill holes. The table shows the number of drill holes, the number of clusters, the cluster distance used, the runtime t for the system components, and the total runtime. As a baseline to compare the total runtime of the algorithm by Suiker et al. [15] is provided where computation was feasible. **System Setup:** All experiments were performed on a workstation equipped with an Intel Core i9-14900K CPU (3.2 GHz, 32 cores), an NVIDIA RTX 4070 GPU (12 GB), and 64 GB of DDR5 RAM.

Experiment	1	2	3	4.1	4.2	5	6	7	8
Drill holes	5	39	34	38	38	6	22	100	200
Cluster	5	13	14	15	13	5	5	77	75
Cluster Distance (m)	6	6	6	4	6	6	6	6	6
$t_{Clustering}$ (s)	1.895	1.633	1.785	1.650	1.655	1.950	4.611	9.223	10.552
$t_{PoseEstimation}$ (s)	85.395	332.579	45.283	409.451	364.005	363.120	83.681	3514.175	3448.550
t_{MPOS} (ms)	0.111	16.646	2.961	0.609	0.299	2.031	0.161	19.669	15.306
t_{total} (s)	90.322	355.242	47.071	411.057	366.522	368.679	100.424	3523.418	3459.117
$t_{total[15]}$ (s)	566.877	-	-	-	-	7438.92	-	-	-

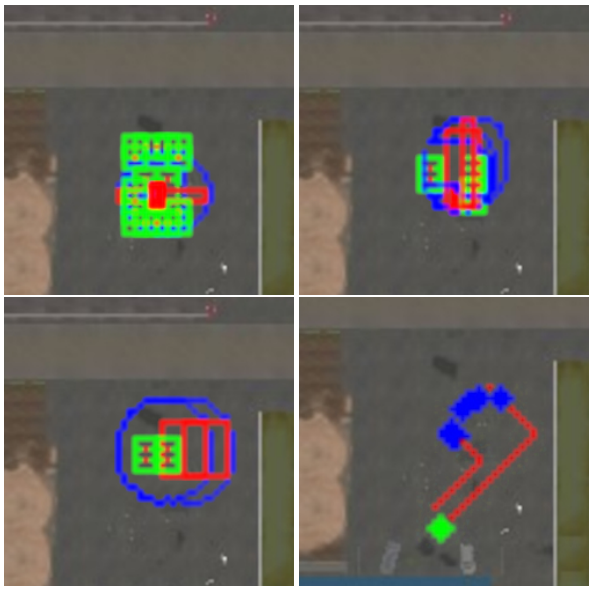


Figure 7. The results of the third experiment. The first three images show the computed largest rectangles (red), the *GCAM* contour (blue), and the *GCAM* holes (green), and the fourth image shows the routing results.

gorithm does compute a good result and takes even small obstacles and holes into consideration when placing the rectangle. Furthermore, none of the computed rectangles intersect any of their drill holes or have an area smaller than the robot's undercarriage. When comparing the computed poses of the robot and their drill order, it becomes apparent that the approach does not place poses between already-drilled holes. This prevents the robot from planning paths through holes, which are not drivable after drilling, as they could collapse or already be filled with concrete or anchors. To experimentally test if the robot can get stuck by drilling holes around itself, a set of points, seen in figure 7, is constructed. This set consists of points spaced two meters apart, and in the middle of the set, a hole of 4×8 meters is placed. This hole is big enough to place the robot inside the point set. When clustering those drill holes, two groups were created, resulting in two *GCAM*s. One being inside the point set and the other surrounding the drill holes. The rectangles calculated based on the *GCAM*s don't intersect any drill holes. None of those poses were fully surrounded by drill spots, thus preventing the robot from getting stuck. The experiments showed that the approach delivers good results for ordering tasks, clustering drill holes, and computing poses of the robot. The time required to calculate the poses was significantly reduced compared to the solution of Suiker et al., bringing computation time per pose down from minutes to seconds. The computation of the vehicle rooting is also fast, with computation times of only a few milliseconds, as shown in table 2. While the computation time compared to the nearest neighbors heuristic increases, its path length is shorter. Compared to the original version of the MPSO algorithm, both path length and computation time are reduced significantly, as shown in Figure 3. Clustering of the drill holes proved to be the most time-consuming part of the system, with the time varying depending on the maximum distance threshold, ranging from a few seconds to minutes. Furthermore, the autonomous movement of the drill arm and the positioning of the robot were tested in a real-world scenario. In this scenario, the drill holes were precisely located and marked using a GNSS system with RTK correction. The robot was able to approach the marked positions and place the drill at them.

6 Conclusion

This paper presented an approach to terrain-dependent analysis for task planning of autonomous drilling excavators. Using information on the work area, drill spots, and the *IRM*, the algorithm clusters the drill holes in groups based on their distance using Voronoi diagrams. For each group of drill holes, the position es-

timation approach computes the positions at which the robot can be placed. The largest polygon inside the contour of the *GCAM* is calculated, and the pose of the robot is determined. For each of the computed poses, a map is created using the results of a wavefront algorithm. An adjacency matrix is created to compute the shortest paths between the poses and their lengths. For routing, a modified particle swarm optimization algorithm was chosen. The result is the order in which the robot travels to the drill holes. The system was tested and verified with special focus on clustering and ordering of the points, yielding promising results for groups with only a few drill holes. In the future, to ensure working with concave polygons is feasible, higher resolutions of the *GCAM* need to be considered while keeping similar computation times. Additional metrics, such as the distance to pit walls, should influence the order in which groups close to each other are processed, and collaboration with other construction-site vehicles or multiple drills should be considered.

References

[1] Garo Hovnanian, Adi Kumar, and Ryan Luby. Will a labor crunch derail plans to upgrade us infrastructure?, 2022. URL `https://www.mckinsey.com/industries/public-sector/our-insights/will-a-labor-crunch-derail-plans-to-upgrade-us-infrastructure/`.

[2] Kelen C Teixeira Vivaldini. Automatic routing of forklift robots in warehouse applications. Citeseer, 2009.

[3] Hardi Kurnianto and Pranoto Hidaya Rusmin. Task allocation and path planning method for multi-autonomous forklift navigation. In *2022 5th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, pages 631–636. IEEE, 2022.

[4] Jongwon Seo, Seungsoo Lee, Jeonghwan Kim, and Sung-Keun Kim. Task planner design for an automated excavation system. *Automation in Construction*, 20(7):954–966, 2011. ISSN 0926-5805. doi:https://doi.org/10.1016/j.autcon.2011.03.013.

[5] Lorenzo Terenzi and Marco Hutter. Toward autonomous excavation planning. *IEEE Transactions on Field Robotics*, 1:292–317, 2024. doi:10.1109/TFR.2024.3485037.

[6] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 109–116, 2006.

[7] Heidi Koivistoinen, Minna Ruuska, and Tapio Elomaa. A voronoi diagram approach to autonomous clustering. In *Discovery Science*, pages 149–160, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-46493-8.

[8] Irwin Sobel. An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*, 02 2014.

[9] Ruben Molano, José Carlos Sancho Núñez, Mar Ávila, Pablo Rodríguez, and Andres Caro. Obtaining the user-defined polygons inside a closed contour with holes. *The Visual Computer*, 40, 12 2023. doi:10.1007/s00371-023-03170-9.

[10] M. Bellmore and G. L. Nemhauser. The traveling salesman problem: A survey. *Operations Research*, 16(3):538–558, 1968. ISSN 0030364X, 15265463. URL `http://www.jstor.org/stable/168581`.

[11] Merrill M Flood. The traveling-salesman problem. *Operations Research*, 4(1), 1956.

[12] Majid Yousefikhoshbakht. Solving the traveling salesman problem: a modified metaheuristic algorithm. *Complexity*, 2021(1):6668345, 2021.

[13] Michael Soulignac, Patrick Taillibert, and Michel Rueher. Adapting the wavefront expansion in presence of strong currents. In *2008 IEEE International Conference on Robotics and Automation*, pages 1352–1358, 2008. doi:10.1109/ROBOT.2008.4543391.

[14] S. Quinlan and O. Khatib. Elastic bands: connecting path planning and control. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pages 802–807 vol.2, 1993. doi:10.1109/ROBOT.1993.291936.

[15] Marcel Suiker, Jörg Husemann, and Karsten Berns. Pose estimation of an autonomous drilling excavator. In *New Advances in Mechanisms, Mechanical Transmissions and Robotics*, pages 191–198, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-87537-3.

[16] Max Reichardt, Tobias Föhst, and Karsten Berns. Introducing finroc: A convenient real-time framework for robotics based on a systematic design approach. Technical report, Technical University of Kaiserslautern, 2012.

[17] Patrick Wolf, Tobias Groll, Steffen Hemer, and Karsten Berns. Evolution of robotic simulators: Using ue 4 to enable real-world quality testing of complex autonomous robots in unstructured environments. In *SIMULTECH*, pages 271–278, 2020. doi:10.5220/0009911502710278.