IPAR 4.0 Kutatási és Innovációs Kiválósági Központ
A projekt azonosítója: GINOP-2.3.2-15-2016-00002
A projekt megvalósulása: 9026 Győr, Egyetem tér 1.
A projekt weblapja: www.ipar40kutatas.hu

Dynamic AGV routing

Author(s): Balázs Csutak
Supervisor(s): Gábor Szederkényi and Tamás Péni
Date: 12th November 2018

Document versions:

Date	Modified by	Description

A projekt megvalósítója: MTA SZTAKI
Székhely: 1111 Budapest, Kende utca 13-17.
Telefon: +36 1 279 6000
Weblap: www.sztaki.mta.hu

Contents

List of Figures

1 Introduction

Optimal route planning based on transport demands is an intensively investigated topic in engineering fields. Depending on the applied model and assumptions, the computational complexity of such task moves on a wide scale. Route planning problems are commonly modeled as optimization problems, which can indeed give us an optimal solution, but scale badly as the size of the map or the number of agents increases. Based on this, the aim of our research is the investigation and improvement of algorithms, which can eventually give a suboptimal solution, but are computationally more efficient than single-step optimization approaches.

Vehicle routing problems have been extensively studied recently in the past years. There are several works dicussing routing in large networks, where the size of the vehicles is relatively small, and thus relation between them is not taken into account. These papers mainly consider dispatching the transportation requests to vehicles, and does not focus on the route computation itself.

In this paper, we investigate optimal route planning for multiple type of automated guided vehicles in a microscopic routing environment, where the size of the vehicles in the system is comparable to the size of the undelying network. For this reason, the route planning algorithm should be prepared to avoid collisions and handle congestion and even deadlock problems. This type of vehicle routing has extensive literature, starting from optimization approaches reaching optimal solution in certain cases to suboptimal systems giving real-time solutions.

The authors of [4] model the problem as a mixed integer optimization, and present a system capable to calculate a set of truely optimal routes. Moreover, they introduce a new Lagrangian coordination and decomposition technique, resolving the problem through disrtibuted calculation and repetitive data exchange between the agents. This way, they use a simple Dijkstra algorithm for individual route planning for the agents, but introduce a more complicated cost function to take into consideration vehicle interdependencies.

In our work, we follow the concept introduced in [1, 2, 3], for resolving conflict between vehicles at the time of route planning. First mentioned in [3], this approach uses time windows and resource reservation to ensure, that all routes planned are conflict-free by design. The idea is continued in [2], where computational complexity of such solution is examined. It turns out, that this system can resolve route planning for individual agents in polynomial time (regarding the number of time windows), and thus, it is capable of online, real time planning. Moreover, in [1], the system is compared to the results achieved by a static routing algorithm (ie. planning routes without care to vehicle relations, and resolving conflicts as they arise), and turns out to be more efficient than traditional approaches.

As for the optimization, we are trying to find a solution for two common optimization tasks: the Online Shortest Dynamic Disjoint Path Problem (OSDDPP), and the Online Quickest Disjoint Path Problem (OQDPP). Basically, in the OSDDPP we are trying to minimize the sum of the time of the vehicles moving along the paths, while in case of OQDPP the algoritm should find the set of dynamic routes resulting in the shortest overall makespan. In practice however, the same suboptimal algorithm turns out to be suitable for both cases.

The research is motivated by its possible applications in Automated Giuded Vehicle (AGV) routing systems. The doctoral thesis, from which this paper starts, has the underlying method implemented in HHLA Container Terminal Altenwerder ©, while this work is mostly aimed for industrial application in a factory cell in Gyor.

In the first part of the paper overview of the related literature is presented, with emphasis on their suboptimal solutions to online route planning. Next, we give a formal, rigorous problem statement, mainly based on the work of Björn Stenzel [1], followed by a detailed analysis of his solution to the problem.

In the second part, we present the simulation framework used for testing and validating our results. Design and implementation of such system, capable of simultaneously handling computation for multiple vehicle types and three dimensional visualization was an important part of our research, as is planned

to be used in our future works as well.

Finally, in the last part we present our test results regarding the performance of the whole route planning system.

2 Detailed task description

To reach the aims we set in the previos section, completition of the following tasks was needed.

1. Literature overview:
   - Overview of literature related to dynamic route planning
   - Overview of literature related to optimalization-based planning
2. Implementation of a simple simulation framework in MATLAB
   - Modeling of factory floorplans as a graph
   - Creating a simple model of the AGV's using movement primitives
   - Simulating the AGVs' behavior
   - Visualization of the simulated space and vehicles
3. Implementation and performance analysis of the algorithms chosen from literature
   - Thorough examination of Stenzel's algorithm
   - Modeling time windows and resource allocations
   - Implementation of the route planning algorithm
4. Extension of the simulation framework to handle multiple AGV types
   - Introduction of aerial vehicles (modeling new movement primitives)
5. Extension of the model to three dimensional graphs
6. Extension of the simulation framework to support three-dimensional visualization
   - Loading and presenting 3D AGV models from `.stl` files
   - Designing factory layouts for demonstration purposes
7. Performance analysis of the algorithms in the 3D environment
8. Documentation and presentation of this work

3 Dynamic routing

3.1 Formal problem statement

To present the problem in a formal, mathematically precise form, following the author of [1], we model the routing environment with a graph $G = (V, E)$ with nodes $V = \{1, 2, ..., N\}$ and edges $E = \{(v_1, v_2, l)|1 \leq v_1, v_2 \leq N, l \in R\}$. The graph is directed, and has no multiple or loop edges. The weight of the edges (representing length of this edge) is denoted by l. Agents can have different traversal time, based on their maximal speeds.

Transportation tasks are continuously arriving for the agents, and are assigned to the vehicles by a higher level dispatching system. Although modeling this system is not part of the route planning problem, in the implementation part we present two rather simple solutions for testing purposes. Formally, we define these requests as follows:

Definition. A *request* is a tuple $r = (s, t, \theta)$, where s is the source node (from where the agent should start), t is the target node, and θ is the earliest time, when execution of the requests can begin.

During this work, without loss of generality, we assume, that this request is assigned to a vehicle already in s by the dispathcing system (in real conditions, traversal of the vehicle to the source node of a transportation task can be viewed as a separate request). For this reason, route planning is done between nodes s and t, by using free time windows after time point θ.

Now, when an agent in s is assigned a request, the aim of the route planning algorithm is finding a dynamic route for it, which fulfills the criterias stated (see definitions below).

Definition. A *dynamic path* in a graph G is defined as a sequence

$$P = (\theta_0, (v_1, \theta_1), ..., (v_k, \theta_k))$$

of $v_1, ...v_k$ nodes and $\theta_1, ..., \theta_k$ timestamps, timestamp θ_i representing the earliest time when node v_i can be entered.

It can be clearly seen, that an agent can follow such path by travelling through the edges between the respective nodes, and waiting on the edge, when they would reach the next node earlier than the timestamp belonging to it. It must be noted, that agents are allowed to wait on edges only (or practically, travel them with lower speed than possible), but they must leave nodes of the graph immediately as they arrive.

The key point behind this concept is, that collision and deadlock avoidance can be realised centrally, by giving disjoint routes to the different agents, while the task of our algorithm boils down to the computation of a set of such routes.

Definition. Considering a dynamic path P in a graph G, the timestamp θ_i is called a *reservation* of node v_i, and the interval $(\theta_{i-1}\theta_i)$ is called a *reservation* of the edge between v_{i-1} and v_i.

Definition. Two dynamic paths are considered *disjoint* if there are no overlapping time intervals between reservation times of the contained edges. Mathematically,

$$P_1 = (\theta_0^{(1)}, (v_1^{(1)}, \theta_1^{(1)}), ..., (v_k^{(1)}, \theta_k)^{(1)}) \qquad P_2 = (\theta_0^{(2)}, (v_1^{(2)}, \theta_1^{(2)}), ..., (v_l^{(2)}, \theta_l)^{(2)})$$

$$P_1 \text{ and } P_2 \text{ are disjoint iff } \forall i < k, j < l : v_i = v_j \text{and} v_{i+1} = v_{j+1} \Rightarrow [\theta_i, \theta_{i+1}] \cap [\theta_i, \theta_{i+1}] = \varnothing$$

Now, that the proper operation is ensured by creating disjoint routes, defining optimisation objectives follows.

Definition. The *duration* of a dynamic path is defined as $\Delta p = \theta_k - \theta_0$.

The first problem focuses on efficiency of utilizing the agents in the system, that is, minimizing the time they spend (or, the route, they travel) during the completition of a given set of requests. While for a set of requests known prior to the route planning this can be modeled and solved as a mixed integer problem, for requests arriving continuously, it can not be guaranteed, that any algorithm can produce an optimal set of dynamic routes.

Definition. The *Online Shortest Dynamic Disjoint Path Problem* is defined as follows:
Being given a sequence of requests $(s_i, t_i, \theta_i), i = 1..k$ find a sequence of disjoint paths $P_1, ..., P_n$, for which $\sum \Delta p_i$ is minimal.

The second one focuses mainly on the time efficiency of the routing system, having the aim to complete as soon as possible all known requests.

Definition. The *Online Quickest Disjoint Path Problem* is defined as follows:
Being given a sequence of requests $(s_i, t_i, \theta_i), i = 1..k$ find a sequence of disjoint paths $P_1, ..., P_n$ with minimal maximum completition time over all paths (so that $\max_{i=1...n} \theta_i$ is minimal)

Again, this optimisation goal cannot be achieved for the continuously arrived requests, but the algorithms discussed are able to find a solution close to optimal.

3.2 Stenzel's routing algorithm

The algorithm itself is kind of a greedy approach. While the task - both in case of OQDPP and OSDDPP - is minimizing the overall cost of the system, the algorithm focuses on minimizing route completition time for the individual agents. As a result, the algorithm boils down to having a number of agents, looking separately for routes optimal for them.

It can be easily seen, that this selfish behavior could potentially lead to countless problems, like deadlocks forming, or agents trying to use the same route at the same time, resulting in time-consuming waiting. To overcome this issue, the algorithm introduces the concept of time windows, aka reservation of nodes and edges of the graph for given time periods. From this point on, every agent planning a route is obliged to respect former reservations, and calculate their route in a manner that it does not disturb the already calculated routes of fellow agents.

This route planning happens iteratively, an order being determined between the agents. This order can consider priority differences between the requests, might be based on how much time is given for the completition of the requests mapped to the agents, or can be chosen simply the order in which the requests arrived. Whichever strategy is chosed is the responsibility of the higher level management system, the route planning ensuring just collision avoidance and optimal solution for the individual agents.

These individual optomalisation goal can be formulated as follows:

Definition. The *Quickest Path Problem with Time Windows* is defined as follows: Being given a graph $G = (V, E)$, a set of time windows for the edges, a request $r = (s, t, \theta)$ and an agent in s, compute a dynamic path with minimal completition time p that uses the edged of the graphs in the free time windows.

The original work also defined the *Shortest Path Problem with Time Windows*, that is, finding a route with the above conditions that has minimal length (sum of the edge costs) and respects the time windows already set. However, as this problem is proven to be NP-hard, it is computationally infeasible for our application. For this reason, both when trying to solve the OQDPP and the OSDDPP, we compute routes that are solution to the Quickest Path Problem.

As formulated by Stenzel, the iterative algorithm works as follows:

Algorithm 1: Iterative routing scheme

Data: Graph $G = (V, E)$, set of requests $R = (s_i, t_i, \theta_i)$ dispatched to agents
Result: Set of dynamic paths P_i serving the requests

```
1 begin
2     foreach request r_i ∈ R do
3         Compute a dynamic path resolving the Quickest Path with Time Windows problem;
          /* Execute Algorithm 2                                                    */
4         Modify time windows to include the new reservations /* Execute Algorithm 3 for the
              given dynamic path                                                     */
5     end
6 end
```

While not giving an optimal solution, this concept has several advantages. First of all, it is computationally feasible, even for large graphs and numerous agents - for details, see derivation of complexity theorems presented in [1]. Moreover, this ensures continuous online computation - transportation requests for agents can arrive continuously, in a realistic manner - each agent getting a new task when the previous one is finished.

Apart from the time window concept, the algorithm is basically a modified Dijkstra route planning for the individual participants.

3.3 Resource allocation using time windows

To formally describe the algorithm, we define the concept of resources, time windows and labels.

Definition. A *resource* is part of the graph, which can be used simultaneously by only a single agent. This can be a node, an edge, or even a set of both.

Since the graph is directed, a resource typically consists of two edges (back and forth) between the same two nodes. By the introduction of the planner graph in 3.5 part to consider the time required by vehicles to turn in the respective graph nodes, this concept becomes even more complex (for instance, the vitrual edges representing rotation in the same physical node are being treated as a single resource) (detailed description on this follows in 3.5)

Definition. A *time window* on a given resource is defined as a pair of time values (a_i, b_i), between which the resource can be freely used by an agent.

It can be easily seen, that a set of time windows completely describes the availability of a resource, while finding whether the resource is free for a particular time interval has logarithmic complexity regarding the number of windows on it. More precisely, we define the reservations of a resource as follows:

Definition. The *reservation* of a resource is given by a set of consecutive time windows:

$$\mathcal{F} = \{(a_i, b_i) | i \in \mathbb{N}\}, \text{where } \forall\ i < j : b_i < a_j$$

Note. The b_i element of the last time window in the set is always equal to $+\infty$, except the case when an edge is permanently reserved due to operation failure, eg. vehicle breakdown completely blocking the edge.

Now, following this concept, making a new reservation on an edge can be defined:

Definition. *Making a reservation* on a resource for an interval (a, b) means finding a time window $(a_i, b_i) \in \mathcal{F}$, for which $a_i \leq a$ and $b \leq b_i$, and modifying it by substracting (a, b):

$$(a_i, b_i) \longrightarrow (a_i, a) \cup (b, b_i)$$

As the $\mathcal{F}$ can contain time intervals only, the new interval is added to the end:

$$\mathcal{F} = \{(a_1, b_1), ..., (a_i, a), (a_{i+1}, b_{i+1}), ..., (a_N, b_N), (b, b_i)\}$$

To keep finding an interval computationally easy, in the implementation the elements of the set are rearranged.

As we already stated, the base algorithm itself resembles Dijkstra's algorithm, which operates by assigning distace values to the edges (the distance from the source node, known at a certain stage of the algorithm), and updating these values based on that stored in the neighbour nodes. Now, to implement this behavior, but taking in consideration the arrival time rather than distance, we introduce the concept of labels.

Definition. A *label* is defined as a tuple $L = (s, t, a, b, p)$, and means, that the agent for which route planning is done, can reach the tail of edge between nodes s and t in the time interval (a, b). The p value is the identifier of the edge, from where the agent arrives.

First of all, we should note, that the reason why the t head of the edge is part of the label is, that due to the construction of the algorithm, these labels are assigned to edges instead of nodes. Secondly, the presence of b as last possible time of arrival is necessary, as the agent might be obliged to leave the previous edge from where it would arrive due to a reservation made formerly by another agent to an interval after b.

Another important modification to the Dijkstra algorithm is, that an edge might be assigned not only a single label, but multiple labels as well. This construction is necessary, as contrary to the simple static route planning, when always the route with lowest distance is certainly the best, and discarding the higher values can be done without problem, here we can not define an obvious order between the labels to choose which one to keep. For instance, due to reservations on the edges, it can happen, that a label with highest arrival value will result in a quicker route after reaching the target.

Finally, we define a relation between labels, so that we can discard those, that certainly result in worse routes than an another label already present.

Definition. Label $L_1 = (s_1, t_1, a_1, b_1)$ dominates a label $L_2 = (s_2, t_2, a_2, b_2)$, if it is a subset of it: $a_1 \leq a_2$ and $b_2 \leq b_1$.

Below, in the discussion of the route calculation, ww explain in details how and why this domination can be used to get rid of unnecessary cases.

3.4 Route computation

In this part, we describe the operation of the route-planning algorithm.

The algorithm consists of four important parts: initialization, the main loop iterating over edges and labels, the label actualication step (which is part of the loop), and finally, the computation of a dynamic route from the labels and reserving the resources used in it.

Similarly to Dijkstra, or almost any other path-search algorithm, we use a priority queue to keep track of elements (nodes, or labels in this case), that need to be processed. In the initialization step, we push labels related to the source node in the queue, so that the algorithm can begin. Second, in the main loop, in every iteration, we select and pop one of the elements from this queue, and *expand* it, which means we update the stored values in the nodes or edges connected to it. The algorithm terminates, when the queue becomes empty, or when we can decide from the currently expanded element, that the optimal solution is reached - when expanding this element involves label updates based on the label having the target node as its head. Finally, based on the values assigned to the edges of the graph, we calculate a dynamic route and make the reservations needed.

Algorithm 2: Dynamic Route Calculation

Data: Directed graph $G = (V, E)$, source node s, target node t, release time θ, function $\tau(e)$ which gives the time required by the respective AGV to travel the edge e, and a set of time windows $\mathcal{F}(e)$ for the edges

Result: A dynamic route P with $\theta_k \geq \theta$ and solving the Quickest Path Problem with Time Windows

begin
  $H = \emptyset$;
  foreach $e \in E$ **do**
    $\mathcal{L}(e) = \emptyset$
  end
  foreach $e : e.tail = s$ **do**
    $L = (e.tail, e.head, \theta, \infty, nil)$;
    H.insert(L);
    $\mathcal{L}(e)$.insert(L);
  end
  while $H \neq \emptyset$ **do**
    $L = H.pop()$;
    if $L.s = t$ **then**
      break;
    end
    foreach $F = [a, b] \in \mathcal{F}(e)$ **do**
      if $L.b < a$ **then**
        break;
      end
      if $b < L.a$ **then**
        continue
      end
      $t_{in} = \max\{a, L.a\}$; // If $L.a < a$, the agent must wait until a
      $t_{out} = t_{in} + \tau(e)$; // First possible arrival to the end of the edge
      if t_{out} ¡ $L.b$ **then**
        foreach $f : f.tail = L.t$ **do**
          $L' = (f.tail, f.head, t_{out}, b, L)$; // This would be the new label
          foreach $\hat{L} \in \mathcal{L}(f)$ **do**
            if L' *dominates* $\hat{L}$ **then**
              H.erase($\hat{L}$);
              $\mathcal{L}(f)$.erase($\hat{L}$);
            else if $\hat{L}$ *dominates* L' **then**
              continue
          end
          H.insert(L');
          $\mathcal{L}(f)$.insert(L');
        end
      end
    end
  end
  Calculate dynamic route based on labels (Algorithm 4).
end

De algorithm can be described by the following pseudo-code, explained below in details.

For the next parts, lets assume we want to plan an optimal route for an agent from node s to node t, respecting the already present resource reservations.

Initialization

First, we initialize the priority queue H as an empty queue of labels, the ordering relation being the comparision between the a values of the elements. Similarly, we assign an empty list of labels to all edges, noted by $\mathcal{L}(e)$.

Then, we look for edges e having the source node as their tail, and insert label $L = (e.tail, e.head, \theta, \infty, nil)$ into H and into $\mathcal{L}(e)$ (θ is the earliest time the agent can start execution of the request, while nil represents, that these edges have no predecessor, from where the agent arrived).

Main loop

Now, in every iteration of the main loop, the label L with minimal arrival time a is popped from the queue. The algorithm checks, whether the target is reached (that is, $L.s = t$), and moves to the computation of the dynamic route if it is (lines 13-15).

If not, similarly to the Dijkstra algorithm, actualization of the consequtive edges takes place. With the loop in lines 16-38, for each empty time window on edge $e = (L.s, L.t)$, traversal possibilities are checked.

If the last possible arrival time to the tail of the edge ($L.b$) is earlier than the beginning of the time window (a), the agent can not pass the edge using this time window. As time windows are ordered in $\mathcal{F}(e)$, no further iteration is required, the algorithm continues by chosing the next label from H. (lines 17-19).

If the first possible arrival time is later than the end of the time window, the agent can not travel the edge using this window. The algorithm is continued with the next time window from $\mathcal{F}(e)$. (line 20-22).

If neither of the above conditions broke the execution of this loop, the agent might be able to travel the edge. Based on the current time window and the traversal speed of the agent, first and last possible arrival times to the end are calculated. The first time the agent can begin traversal of the edge is $t_{in} = \max\{a, L.a\}$: if beginning of the time window $L.a$ is later than the first possible arrival time a, the agent waits on the previous edge, and enters at $L.a$ only. Consequently, the first possible arrival time to the head of the edge (or, the tail of the next one) is $t_out = t_{in} + \tau(e)$. If the agent can travel the edge until the end of the time window (meaning that $t_{out} < b$), labels on all edges f going out from node $L.t$ are actualized based on domination rules (line 25-37).

Label actualization

Actualization of the labels happens by creating the new label based on first and last possible arrival times ($L' = (f.tail, f.head, t_{out}, b, L)$), then whether it is dominated by other labels or contrary, it dominates some others. It can be clearly seen, that if a label is dominated by any other label, we should not take it into consideration any more, as using the label dominant label completely covers the possibilities introduced by the another one.

For this reason, all labels of all successor edges are checked (nested `foreach` loop pair in lines 26-36. If the new label is dominated by any of the already present ones, it is not inserted in the queue, and no further checks are done: the algorithm continues by examining the next successor edge. Otherwise, the new label is inserted in H and in $\mathcal{L}(f)$ as well. During this process, one more check is done: if the new label dominates any of the already present ones, that label is removed from H, and from $\mathcal{L}(f)$ is well, as the new label will completely take its role.

Route calculation

When the algorithm pops a label from H with its tail being equal to the goal node, the algorithm stops. Now, by stepping backwards based on the label, a dynamic route is generated.

Algorithm 4: Calculate dynamic route from labels

Data: Directed graph $G = (V, E)$, starting label $L = (s, t, a, b, p)$, $L.s = t$
Result: Dynamic path P

```
1  begin
2      k = 1;
       /* Compute dynamic route backwards                                  */
3      while L ≠ nil do
4          v_k = L.t;
5          θ_k = L.a;
6          L = L.p; k = k + 1;
7      end
8      reverse(P);
9      θ_0 = θ_1 − τ(s, v_1);
10 end
```

Modifying time windows according to new reservations

Modification of the time windows to include new reservation is quite straightforward, if a properly constructed dynamic path is given. Assuming that if a wait operation during a path required, agents are instructed to wait as late as possible, the time slots for which an agent occupies the edge (v_i, v_{i+1}) is exactly $[\theta_i, \theta_{i+1}]$.

A more interesting question is, however, which set of time windows to modify to avoid collisions indeed. While reservation of the resource belonging to the edge (v_i, v_{i+1}) comes naturally, some neghbouring edges might also be affected. To prevent any issue coming from this (like agents arriving from different direction to a node with ε time difference colliding), we define the concept of conflicting edges. Now, these edges should not be checked to be free in the step of dynamic route calculation, they should be reserved when adjusting the time windows according to the dynamic path though. In our model, when reserving an edge, the set of conflicting edges we reserve are all those connected to v_i or v_{i+1}.

Based on this, the implementation of the algorithm can be described by the following pseudo-code:

Algorithm 3: Modifying time windows according to new reservations

Data: Directed graph $G = (V, E)$, dynamic path $P = (\theta_0, (v_1, \theta_1), ..., (v_l, \theta_l))\}$, a set of time windows $\mathcal{F}(e)$ for the edges, set of conflicting edges $confl(e) \forall e \in P$
Result: A new set of time windows $\mathcal{F}(e)$ including the reservations for P

```
1  begin
2      foreach e = (v_k, v_k + 1) ∈ P do
3          foreach f ∈ confl(e) do
4              foreach F_i = [a_i, b_i] ∈ F(f) do
5                  if θ_{k+1} <≤ a_i then
6                      continue
7                  end
8                  if a_i ≤ θ_k and θ_{k+1} ≤ θ_{k+1} then
9                      for m = end(F(f)); m >= i + 1; m = m − 1 do
10                         F_m = F_{m−1};
11                     end
12                     F_i = [a_i, θ_k];
13                     F_{i+1} = [θ_{k+1}, b_i];
14                 end
15             end
16         end
17     end
18 end
```

In this code, the algorithm iterates through all edges in the dynamic path (loop between lines 2-17), and makes a reservation for all conflicting edges (foreach loop in lines 3-16). The reservation is made by finding the time window in which the agent travels the edge (line 8) and splitting the respective time window in two parts. To keep the set $\mathcal{F}(f)$ ordered, elements of the set are shifted (lines 9-11), and the new window is inserted right after the one being splitted (lines 12-13).

3.5 Modeling AGV movements and the routing environment

As the last part of the dynamic routing algorithm, a model of AGV movements, and the corresponding representation of the environment is presented.

First of all, the factory floorplan is handled a a simple directed graph, with edges representing the paths an agent can follow, and nodes being the intersections between such paths. In the further discussions, this graph is called the *factory graph*.

However, as real life agents are not capable of arbitrary movements (for instance, can not change their direction in zero time), we use some predifened movement primitives to decribe their behavior. These primitives are `GO_STRAIGHT`, `TURN` and `WAIT`, and we assume, that all agents in the system can execute any of them.

Basically, the `GO_STRAIGHT` stands for straight, horizontal movement, in the main travelling direction of an agent. If no such direction exists (for instance, a quadcopter can travel equally in all four directions), one is chosen at the beginning of the simulation. The primitive has one parameter, the distance the agent is supposed to travel. To be able to handle aerial vehicles as simple AGVs, an optional second parameter, the vertical movement can be added. This value means, that the AGV (provided it is an aerial vehicle capable to do so) will rise/sink this distance during the forward movement.

The another movement primitive `TURN` is rotation in place, i.e changing the forward direction with position of the mass point remaining unchanged. This primitive requires a single parameter, an angle (positive or negative value representing left and right turn respectively), that describes how much the agent should turn. We assume, that the agent can not change its height during a `TURN` operation.

The third movement primitive represents waiting in-place, without changing position or orientation.

Now, while the *factory graph* would be enough to compute routes including `GO_STRAIGHT` and `WAIT` operations, time required for the turning must be somehow included in the weight of the paths. To be able to use the algorithm without modification, a virtual graph called *planner graph* was introduced. For every physical node in the factory graph, we generate a virtual node for every possible direction in which the agent can arrive to or depart from the node. Edges representing rotation are added between neighbouring directions (see fig. 1), edges representing transition are preserved between the virtual nodes having the same direction value.

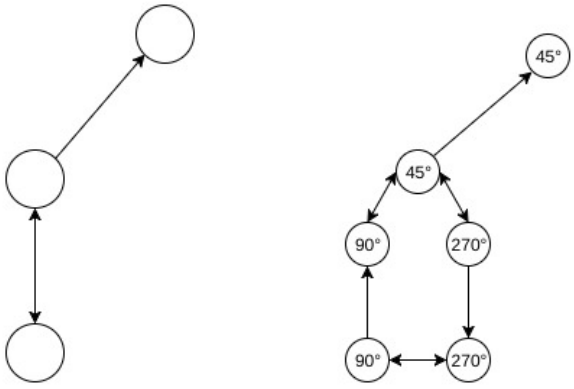


Figure 1: Contruction of a simple 2D planner graph

From this point, running the route planning algorithm on the planner graph would result in dynamic routes, which can be translated directly to a list of movement primitives.

4 Implementation

In this part we present the implementation of the upmentioned algorithms, created for testing and validating their performance. The simulation was realised entirely in MATLAB, and is formed by two main components. First, a general framework was created for simulating AGV movement in a three dimensional coordinate system, where all AGVs act based on predefined task-list, given by the user. This framework provides a simple model for dealing with multiple type of AGV vehicles through the class called `AGV`, and features a simulation loop, which iteratively computes and shows the AGV movements in our system (the `Simulation` class). As there is a possibility to throw and handle events during the simulation (like an AGV reaching its destination and waiting for new tasks to be added), it is completely fesible to simulate route planning algoritms in general.

Next, we moved on to impementation of Stenzel's algorithm. A MATLAB model of the graphs (class `FGraph` for the factory, class `PGraph` for the pllanner) and time windows (`Resources` class) was introduced, followed by the implementation of the time-window based route planning (`algorithm 2`) and route reservation (`algorithm 3`)algorithms. As all route-planning algorithms investigated assume a high-level dispatching system giving route sources and destinations to the agents, in the test scripts we also created a simple dispatching model (`demo_gyor.m`). We already mentioned, that the algorithm cannot deal with idle AGVs waiting for a new order (for them, without reservations, these agents disappear from the network), so introduction of so-called parking places was necessary for the simulations. This way, not sending multiple agents to the same parking place (or workstation) becomes the responsability of the dispatching system.

Finally, we present the upcoming implementation challenges, caused by transition to three dimensional modeling.

4.1 The MATLAB framework

AGV modeling (`AGV.m`)

In this framework, all moving AGV agents are modeled as mass points, moving in a coordinate system, and are displayed as 3D CAD models loaded from `.stl` files. They all belong to the same `AGV.m` base class, having the following important properties and functions:

- `position` and `rotation`: a three-element double vector, position of the mass point in the coordinate system and a three by three rotation matrix, describing rotation of the initial 3D model.
- `modelVertices`: n by three matrix, containing the initial position of the vertices of the loaded 3D CAD model. This is centered to the origin of the coordinate system and normalized (all AGV having size 1 along their longest dimension). Multiplied elementwise with the rotation matrix, and added to the position vector, this results in a 3D AGV model in the coordinate system with the right position and orientation.
- `velocity` and `angularVelocity`: velocity of the agents during straight movement and rotation. As all AGVs modelled are capable of waiting in place, this is assumed to be constant without loss of generality.
- `tasklist`: LIFO list of movement primitives, with reqiured parameters (discussed above).
- `update(dt)`: calculates the next state of the agent after a dt timestep, based on the properties described above

Now, by modifying the tasklists using the `addTasks` helper function, users of the framework can assign

tasks to the AGVs, identical to the movement primitives assumed above. This way, any route to be followed can be easily described as a list of these tasks, and simulated by the system.

The most significant part of this modeling being the update function, we discuss that more in details. This function is called periodically by the main simulation loop, and does the following: first, it checks whether the current task (that on the top of the task list) is finished. If not, execution of that is continued, else it pops the next task from the list and begins the execution. As for the execution, the algorithm calculates the next position based on the respective movement primitive, modifies the parameters of the task (to keep track of how much of it is done), and recalculates the position of the model vertices displayed.

For more implementation details, see documentation of the MATLAB code in Appendix 1.1

Main simulation loop (`Simulation.m`)

The simulation is run by the class named `Simulation`. This has a list `AGVs` containing the handles of all agents in the system, and a `time` variable, which contains the time elapsed since the beginning of the simulation. New agents can be added using the function `addAGV`. The simulaiton can be started by calling the `simulation` function, which contains the main simulation loop.

The simulation loop is an infinite loop, which first deretmines the next timestep, and calls the update function of all agents whith that value afterwards. The timestep is obtained by iterating thorugh all agents, and checking how much time their need in order to complete their current task (on the top of their command list). The algorithm chooses the minimum from the set of these values and the default timestep value of 0.04 seconds. This approach ensures, that all AGVs will complete their tasks exactly at the end of a simulation cycle, while also preserving a minimum of 25 updates per second.

All AGVs have the possibility not to return anything, or to throw an event, as the return value of their update function. If such an event is returned, the main simulation loop terminates (all state variables being preserved), and returns control to the caller script. This way, when for instance an agent finishes all of its tasks, it can request new task from the dispatching system. The simulation can be resumed by calling the `simulate` function again.

For documentation and visualization purposes, this funciton also handles how graphics is displayed. It takes in consideration the time required for computation, as well as for the draw operations, when choosing the amount of idle time required to provide a constant frame rate for the viewer. The class can also be used for video construction, by simply passing a recorder object to it in the constructor. This way, all frames are put after another in `.avi` format.

For more implementation details, see documentation of the MATLAB code in Appendix 1.2

4.2 Routing algortithm

Factory graph

The first step to implement the routing algorithm was finding the right representation for the environment, in which the routing takes place. As it does not has an effect on the efficiency of the algorithm, we chose to model our factory floorplan as a simple directed graph, and store the position of the nodes and an adjacency matric.

In the implemented code, this appears as the `FGraph` object. The class has two attributes: the `vertices` vector and the `adjmat` adjacency matrix. The `vertices` has size $N \times 3$ matrix, in which each row contains the (x, y, z) coordinates of a node - thus, the nodes are identified by their index in the row $(1, 2, ..., N)$. The adjacency matrix is a $N \times N$ boolean, and a_{ij} has value `true` if there is an edge between the i-th and j-th node. As the graph is assumed to be directed, this matrix is not necessarily simmetric. It was considered to store the weight of the edges in the adjacency matrix as well, however, due to the construction of the planner graph (see below) it is not required.

The matlab class has four methods to support construction of arbitrary graphs, called `add_vertices`,

`delete_vertices`, `add_edges`, `delete_edges`, all being able to perform the insert and delete operation without breaking the structure already set. The `add_vertices` and `delete_vertices` change the number of nodes, and thus shifts the identifier of some nodes, though.

For more implementation details, see documentation of the MATLAB code in Appendix 2.1

Planner graph

The planner graph is represented by the class `PGraph`, and it is constructed according to the description above (see 3.5). The class has two attributes: the `vertices` vector and the `adjmat` adjacency matrix. The `vertices` has size $N' \times 3$ matrix (N' being the number of virtual nodes), in which each row contains the number of the corresponding phisical node in the planner graph, and an angle value $\phi \in (-pi, pi]$ containing the direction which the virtual node stands for.

The adjacency matrix is a $N' \times N'$ (positive) double matrix, and a_{ij} contains the weight of the edge between virtual nodes i and j. If these nodes belong to the same physical node, this weight is the amound of degrees (in radian) the agent has to turn from one to the another; otherwise, the weight is the length of the physical edge.

While using this representation, turning behavior and the time required for an AGV to turn into the required direction is covered for arbitary graphs without additional computations, it introduces difficulties in resource allocation. It must be noted, that these virtual edges, as well as the virtual nodes represent the same physical resource, and must be treated accordingly when reserving a node or an edge for the algorithm.

For more implementation details, see documentation of the MATLAB code in Appendix 2.2

Resource allogation

As already described above, the route planning algoritms avoids collisions and deadlocks using the concept of resource allocation using time windows, which means, that all physical resource (like a node or an edge) can be reserved by an agent for a given interval of time. Moreover, these phisical resources need to be mapped to the elements of the planner graph, so an algorithm working on that can access resercation data for a virtual edge or node.

To keep the implementation simple, a `Resources` class was created to store this data. The class has a property called `timeWindows`, which is a cell array containing the free time intervals for all resources, and a property `resource_ids` ($N_{planner} \times N_{planner}$, where $N_planner$ is the number of nodes in the planner graph), in which $resource_ids_{i,j}$contains the index of the resource belonging to the edge from the i-th node to the j-th node (when $i \neq j$) or to the node i (when $i = j$). If there is no such resource (there exists no edge between i and j, value 0 is stored).

Generation of the resources happens in the following way: first, we assign values from 1 to $N_{planner}$ to the main diagonal of the $resource\_ids$ matrix. Next, we iterate through all edges in the graph, and calculate how many resources are needed (for instance, bidirectional edges, and edges representing rotations share a common one), and insert the values into the matrix.

Finally, we create the timeWindows cell array with R elements (R is the number of resources needed), and initiate $timeWindows\_i$ as a 1×2 matrix having values 0 and ∞ (which means, that all resources are completely free).

Reservation of resources happens the following way: when the route planning algorithm has to reserve a node or edge, it first looks up the resource id in the `resource_ids` matrix. Next, it iterates through the rows of the matrix stored in the cell $timeWindows_{id}$, and checks whether the interval for which the reservation is needed is the subset of one of the stored interval. Finally, if it is, reservation happens by cutting the interval in two parts. The second interval is inserted right after the first one, shifting the rows of the matrix.

The `Resources` class is also able to create a human readable plot of resource reservation, so working of the algorithm can be visualized in real time. As we can see in figure 2, the resources are represented by horizontal bars, green segments marking the free time windows (the intervals stored in the matrix),

and red segments marking the reservations. It can be clearly seen, that red marks align obliqely, as an agent moves along a path, and reserves consequtive resources for consequtive time intervals.

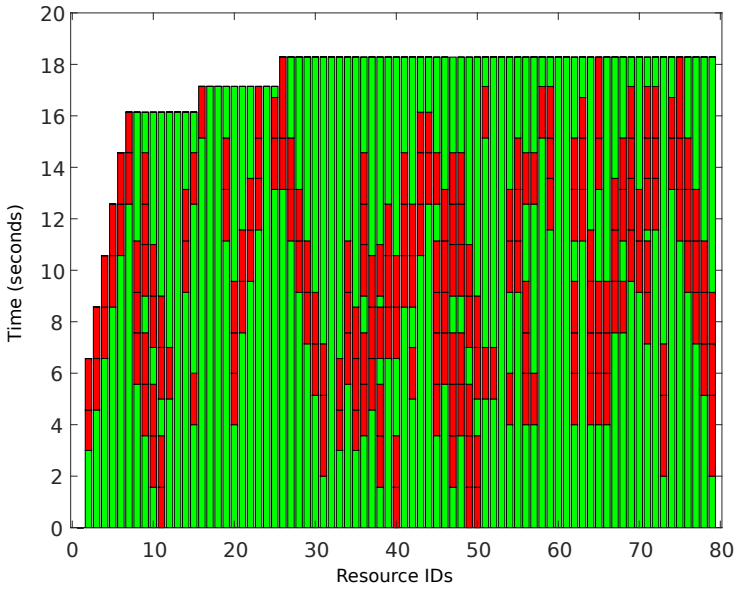


Figure 2: Reservation of resources

Route calculation

Following the route planning method described above, the algorithms for route planning and for reservation of the route found are implemented as functions `algo2` and `algo3` respectively (the naming follows the original names given by Stenzel, the algorithms being almots identical to that described in his paper).

In the initialization part, all edges in the (planner) graph are listed, and a priority queue is created for them, capable of storing labels. For practical reasons, in fact the algorithm creates an array of structures, which contain head and tail info of an edge, as well as the priority queue belonging to it. In the same iteration, if the edges have the starting node as head, a label is added to them, stating that the agent can begin travelling on them at any time beginning with the relase time θ.

For more implementation details, see documentation of the MATLAB code in Appendix 2.4,2.5,2.6.

Parking places

As the route planning algorithms see the agents only through the reservations made by them, it was necessary to remove the AGVs from the graph when not executing a task. To resolve this, parking places were introduced, as nodes aligned at the side of the graph. We assumed, that those parking places are big enough to provide space for any number of agents, or that the dispatching system takes care of not sending too many of them to that particular place.

Now, with this assumption, the only requierment from the dispatching system is, that only routes from one parking place to the another should be planned. This way, during normal operation (without delay or for instance vehicle breakdown) no agent can remain in the graph without the respective resource it uses being reserved for it.

Regarding the implementation, the introduction of these places does not influence at all remaining part of the route planning system.

Dispatching

Although design and implementation of a complex dispatching system was well beyond the scope of this project, creation of a rather simple one for testing purposes was inevitable. For this reason, at first a zero-logic system was created, which for any AGV finishing its task assigned a random target node from the graph.

However, as described above, proper testing of the algorithm required a system capable of handling the parking places. This system was implemented as well, by keeping a list of parking places, and choosing one randomly based on their status. We decided to use two possible states for the parking place: free or occupied. At first, all places from where agents start are set to occupied, the others are set to free. Next, when route planning for agents begin, a parking place is set to occupied as soon as an agent gets a route planned to there. The place, from where the agent starts is set free at the same time, even though the agent might not be able to leave at that moment. The system assigns only free places as destination (choosing randomly), behavior which ensures, that two agents cannot be routed to the same place, colliding there (outside of the graph).

It has to be noted, that the concept presented is far from being efficient, this is not among its purposes though. In real application, where agents have a specific goal, waiting at the other side of the graph just because an another agent is heading to the same destination, is inacceptable waste of time. For testing the route planning, choosing another random target in such situation is perfectly enough.

Extension to 3D models

Due to practical demands, the next step in our project consisted of extending our model and implementation to support multiple types of vehicles, including quadrotors capable of aerial transport. The modification includes several challenges, starting from three dimensional planner graph generation to providing different weights, and even completely different graphs for the individual AGVs in the system. Moreover, the differences between these graphs (inevitable because not all vehicles are designed to travel along air edges) impose the re-implementation of the resource allocation system. Finally, the presence of AGV's with different behavior cannot be simulated without slight modifications to the main simulation class, and the graphical interface either.

Three dimensional planner graph

As discussed above, the planner graph - generated based on the factory floorplan - is a virtual graph, where a node represents the actual position of an intersection and the direction of the vehicle as well. Consequently, rotation time of the vehicles can be taken into account as the weight of the virtual edges, connecting these nodes.

During the creation of our three dimensional model, at first usage of solid angles and an even more complex virtual node generation formula was considered. Later, to keep it simple as possible, we came up with a more natural extension, assuming more simple movement primitives for the aerial vehicles (discussed below in details).

In the process of planner graph generation, we project our three dimensional graph to a plane, resulting in a regular floorplan. In the next step, we generate out planner as discussed above, finally, we assign the same height value to the virtual nodes, as the corresponding physical node had. Now, the only problem remaining are the completely vertical edges: as the projection of the two nodes fall to the same two-dimensional point, it is unclear to which virtual node (which direction value) should these edges connect.

This case was handled in the following way: every node connected to such an edge was assigned a virtual node with direction value 0, and these nodes were connected by the preserved edge. Naturally, if the two physical nodes had another common directions (which is mostly the case in the graphs used), those were connected as well.

The approach discussed completely fits out applied aerial AGV model, which - compared to the regular AGV - features one more movement primitive: during a straight transition, it can increase/decrease its height value as needed. While real-world quadrotors are capable of rotation during a rise/sink operation as well, this behavior was omitted from out system.

Different graphs for vehicles

Introduction of quadrotors in out system was a major modification, as handling of different types of vehicles (with edge travelling capabilities moving on a wide scale) became necessary. To address the issue, the AGV model and the route planning algorithm was modified as follows.

First, the AGV model was changed to include the planner graph, on which the respective AGV can travel. This planner graph can be a clone of the original planner generated from the floorplan (which is the case when an AGV can travel along all edges), or a subgraph of it (edges/nodes unreachable for the AGV are omitted, and weights of the preserved edges can be changed based on the vehicle's properties). Addition of new nodes or graphs is prohibited to ensure consistent resource allocation. As the resources are common for all AGVs, these are generated based on the original planner graph.

The route-planning algorithms undergone a single modification: when expansing an edge label, neighbouring edges and traversal times are queried using the planner model of the AGV, for which the route planning happens. Nevertheless, resource allocation is done based on the original planner graph.

Simulation class and graphics

While changes to the simulation and graphical system are more an implementation issue than research task, these modifications were inevitable to effectively test and validate the algorithm itself in the new situation.

To visualize the AGVs in a three dimensional environment, MATLAB's built-in graphical tools were used. The vehicle shown is a patch object, loaded from an `.stl` file, containing a well-designed representation of the physical AGV's and quadrotors used by the MTA-SZTAKI. Moreover, the graph and the factory outlook was also adopted from the institute, so the vehicles are simulated in real conditions.

The simulation cycle was redesigned in an object-oriented approach, using the features offered by MATLAB. This means, that every AGV object (belonging to the AGV base class) can have its very own 3D model, speed, and behavior, while new typed can be added without touching the already present ones.

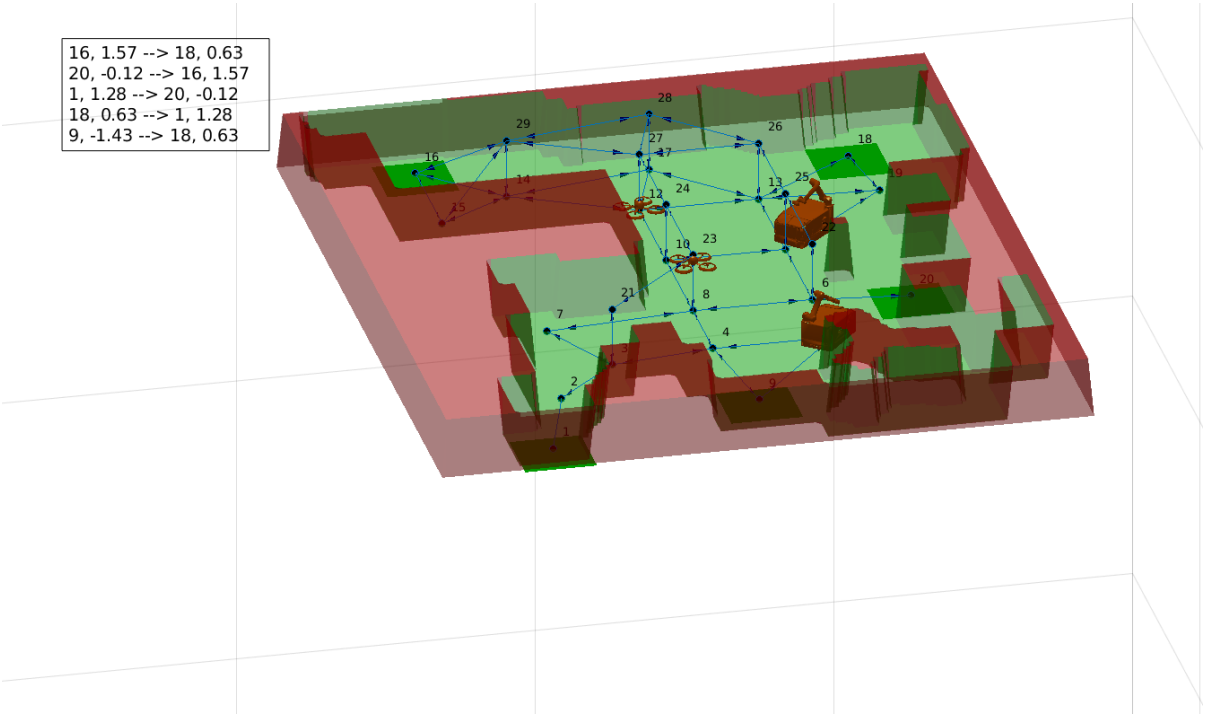


Figure 3: Demonstration of 3D graphics environment

5 Test cases

To extensively test the implemented framework system, and verify the usability of the implemented algorithm in the scenarios needed, an exact model of the factory cell from Gyor was realised in the simulation system. This process involved loading and transforming the data acquired from measurements of the place, so that a three dimensional representation of walls and objects would appear. Next, a directed graph respecting the loaded floorplan was created, followed by the generation of nodes and edges in the air, suitable for quadrocopters only. Finally, after generating a planner graph and nominating some parking places / workstations (nodes, from which and to which transportation requests are arriving), two ground AGVs and two quadcopters were loaded and launched.

In this section we present and explain the simulation results only. For implementation details, see Appendix 3.1.

Loading factory floorplan

As a result of measurements and some preporcessing of data, the following layout was obtained in `.csv` format (figure 4). The file contained a `true` or a `false` value for each coordinate pair $(x, y) \in [-66; +80] \times [-74; +55]$, indicating whether an AGV can go to that place or not.

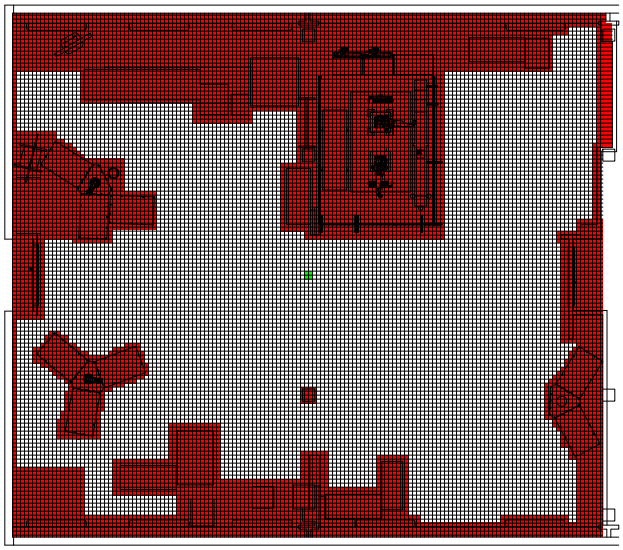


Figure 4: Floorplan of the factory cell located in Gyor

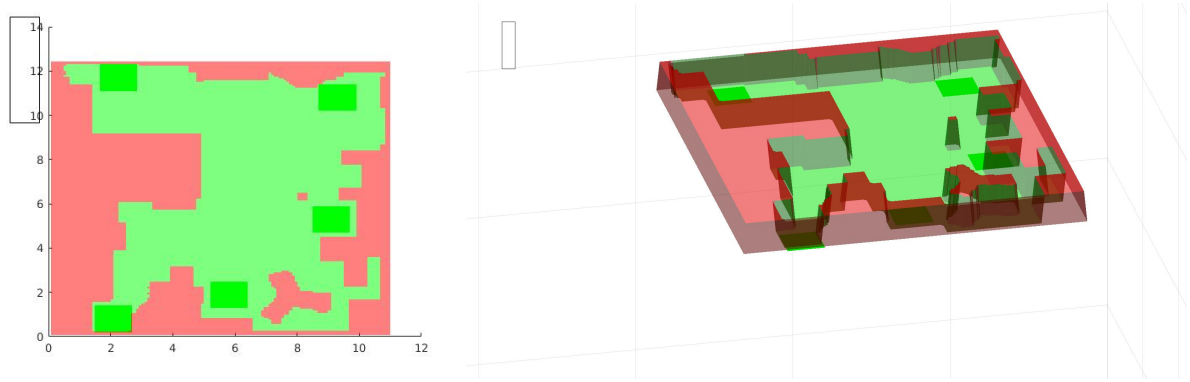
The file was loaded in MATLAB, and the following representation (a surface in a three dimensional space), including walls and blocking objects (figure 5) was created.

In the next step, we realised the *factory graph* containing 18 nodes on the ground, and another 11 ones in the air. By connecting the adjacent noded, a graph with a total of 50 edges was obtained. There were five workstations added, to physical nodes 1, 9, 16, 18, 20 (marked by the green squares on the floor). The graph can be seen on figure 6.

Finally, the planner graph (`PGraph` object) and the resources (`Resources` object) were generated. The planner graph resulting from the above factory graph had 158 vertices and 506 edgeswhile the resources object contains 79 resources (set of time windows).

Loading AGVs and generating requests

The AGVs were loaded to the following positions: two groud AGVs to nodes 9 and 18, and two quadrotors to nodes 1 and 16. In this order, the agents had the targets 18, 16, 9, 20. The route planning took



(a) Layout from above

(b) Layout in perspective

Figure 5: Layout generated by matlab based on the floorplan

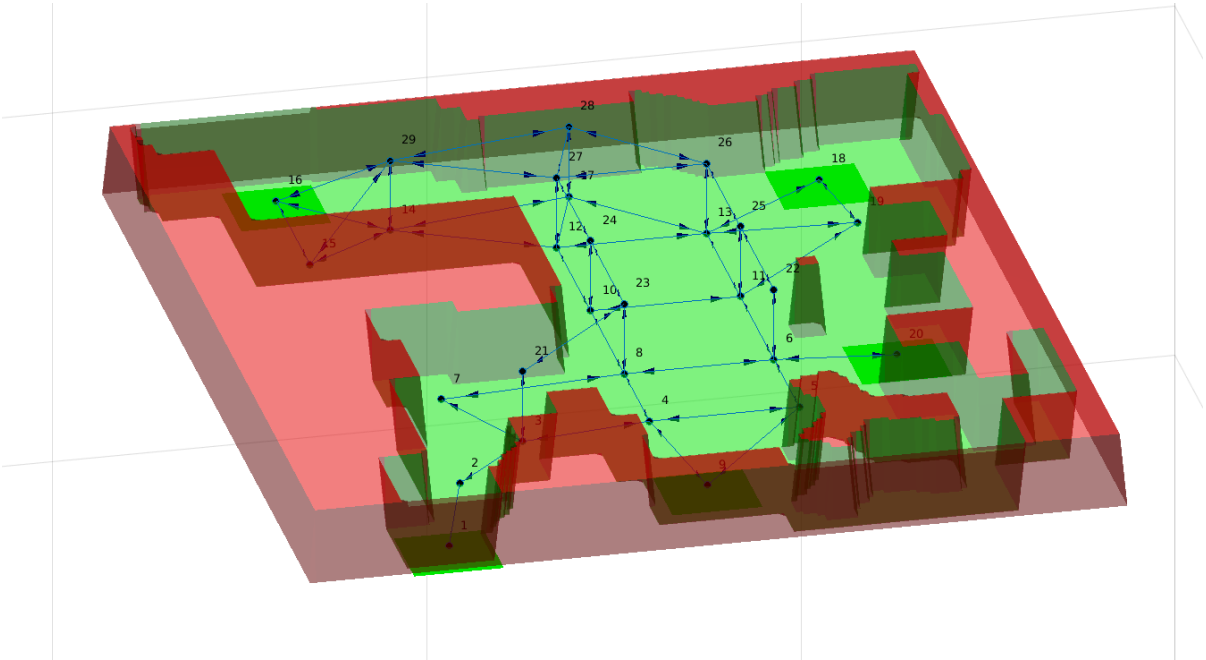


Figure 6: FGraph object of the factory cell

place in order 16, 18, 9, 1 (the numbers being the nodes the agents started from). The routes calculated and followed by the agents can be seen in figure 7. It can be easyly observed, that agent starting from 18 chose route $\{18, 13, 17, 14, 16\}$ instead of the spatially shorter one $\{18, 13, 12, 14, 16\}$ to aviod interference with the route starting from 16 planned before. As for the performance of the algorithm, all route planning operations took place in time less than 0.1 seconds, which could be further reduced by optimalization of the implementation.

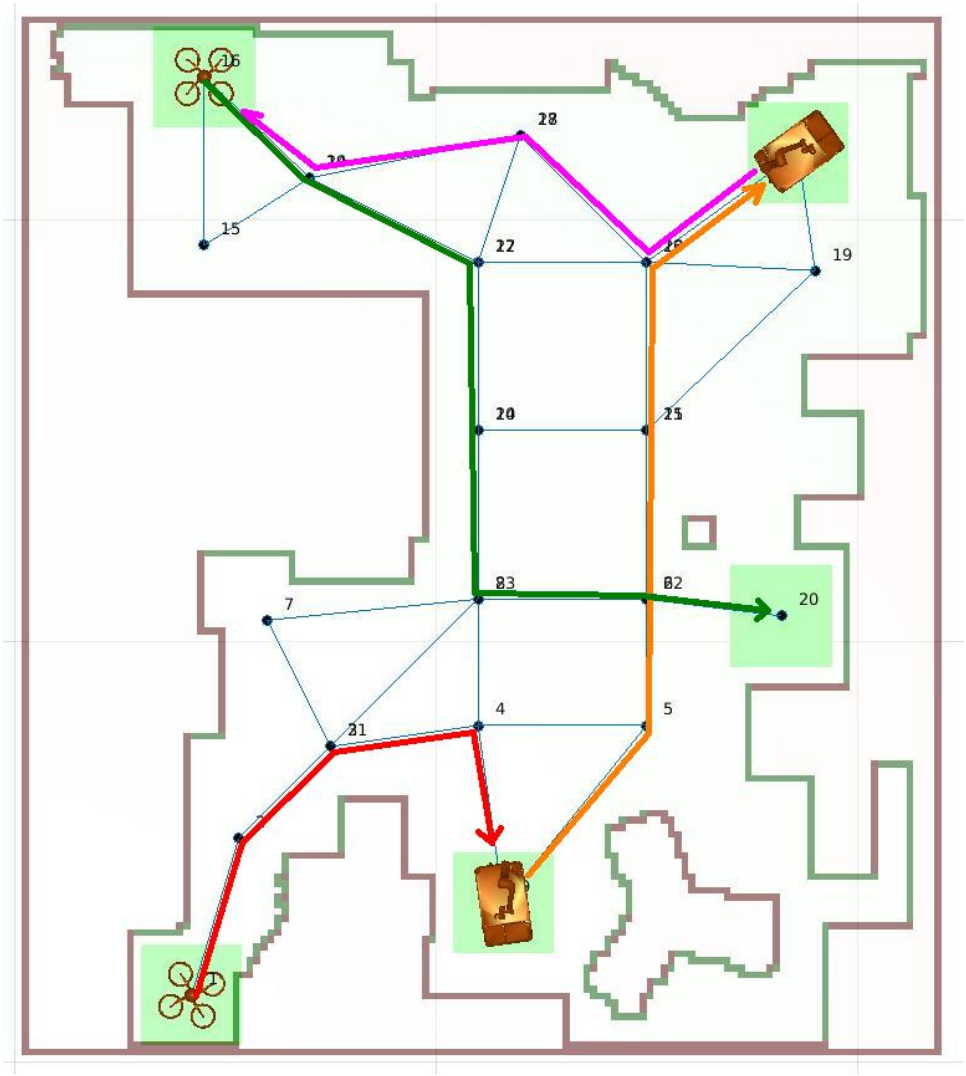


Figure 7: Routes planned using the algorithm

Videos showing the operation of the route planning system, as well as the simulation framework in the factory cell from Gyor, and in a much bigger factory cell containing over a hundred of physical nodes, 200 edges and as much as 10 agents, are available at [6]. The algorithm had a decent performance on those graphs as well (computation times remaining under 1 second in any circumstances).

6 Summary

In this work, we have presented the operating principle and implementation of an algorithm, capable of providing online disjoint autonomous vehicle route planning for multiple type of AGVs moving in a microscopic routing environment.

First of all, thesis of Stenzel [1] was read and carefully examined regarding the operation and implementation of the algorithms, including investigation of how realistic the requirements are. Next, modeling of the agent behaviors took places, by introducing the movement primitives, that each type of AGV is able to execute. By this step, we could create a routing system capable of handling together all agents, and providing route planning for ground and aerial vehicles at the same time. A comprehensive environment model was also created by the definition of planner graphs, which allowed the algorithm to take in consideration time required for all movement primitives, including turning behavior.

Second, the initial MATLAB framework provided at the beginning of this work was extended, to include all the features required. Handling of resource allocation, illustration of time windows, and new AGV movement primitives were implemented. The framework was added a completely new 3D visualization system, and the ability to load and simulate three dimensonal AGV models. Generation of planner graph was reimplemented to include three dimensional factory graphs as well.

In the third step, the algorithm was completely implemented as described in the original paper, and its operation was tested on some factory configurations. To simulate as realistic conditions as possible, a three dimensional model of the factory cell in Gyor was created and loaded, and some test runs were carried out. The algorithm showed excellent performance in all of the scenarios.

Summing up, the problem of online disjoint route planning for this factory cell was solved, but this algorithm hides even more interesting possibilities for the future. Our plans include more subtle resource management and introducing more realistic and efficient algorithms for exceptional cases (like vehicle breakdown). To provide an even more general and formal way for handling these tasks, we are looking forward to the introduction of time-window based temporal logic in the near future.

Bibliography

[1] Stenzel B. (2008), *Online Disjoint Vehicle Routing With Application to AGV Routing*, PhD thesis, Technical University of Berlin, 2008

[2] Gawrilow E., Köhler E., Möhring R., Stenzel B. (2008),*Dynamic Routing of Automated Guided Vehicles in Real-time*. In: Krebs HJ., Jäger W. (eds) Mathematics – Key Technology for the Future. Springer, Berlin, Heidelberg

[3] Möhring R., Köhler E., Gawrilow E., Stenzel B. (2004), *Conflict-free Real-time AGV Routing*, In: Operations Research Proceedings 2004: Selected Papers of the Annual International Conference of the German Operations Research Society (GOR). Jointly Organized with the Netherlands Society for Operations Research (NGB) Tilburg, September 1–3, 2004 (pp.18-24)

[4] Nishi, M. Ando, M. Konishi, "Distributed route planning for multiple mobile robots using an augmented Lagrangian decomposition and coordination technique", Robotics IEEE Transactions on, vol. 21, no. 6, pp. 1191-1200, 2005.

[5] Jrgen Bang-Jensen , Gregory Z. Gutin, Digraphs: Theory, Algorithms and Applications, Springer Publishing Company, Incorporated, 2008

[6] `https://nextcloud.sztaki.hu/index.php/s/Lr943rFRzjmoEp5`

1 Matlab framework

1.1 AGV.m

```
classdef AGV < handle

    %//////////////////////////// PROPERTIES ////////////////////////////////

    % Properties required for graphics
    properties (GetAccess = public, SetAccess = private)
        patch % the 3D patch belonding to the object
        modelVertices % initial vertices of the object
    end

    % state variables of the AGV
    properties (GetAccess = public, SetAccess = private)
        % this must be available somehow, as graphics relies on it
        position % position of the object
        rotation % rotation of the object

        % These properties are used just by the primitive model, and can be
        % replaced if a more complex one is included.
        % The timeToFinishTask() and update() functions must be
        % adjusted accordingly though.
        forwardVector   % direction of the robot in its own coordinate
            system, typically [1 0 0]
        velocity        % speed of movement during GO_STRAIGHT operation
        angularVelocity % speed of movement during TURN operation
    end

    % Properties for task execution -- used by dispatching and routing
    % algorithm.
    properties (GetAccess = public, SetAccess = private)
        currentTask
        tasklist

        planner

        source
        destination
        workstations
    end


    %//////////////////////////// METHODS ////////////////////////////////

    % constructors, copy constructor, etc.
    methods
        function this = AGV(filename, position, rotation, planner,
            workstations)
            % load the stl model from file

            model = stlread(filename);
            model = reducepatch(model,0.05);
```

```
            this.patch = patch(model,'FaceColor', [253, 106, 2]/255, ...
                'EdgeColor',        'none',         ...
                'FaceLighting',     'gouraud',      ...
                'AmbientStrength', 0.15);



            material('metal');

            % center and normalize the model
            offset = (max(this.patch.Vertices) + min(this.patch.Vertices))
                /2;
            this.patch.Vertices = this.patch.Vertices - repmat(offset, [
                size(this.patch.Vertices, 1) 1]);
            this.patch.Vertices = this.patch.Vertices ./ max(max(abs(this.
                patch.Vertices)))/2;


            % save initial vertices for graphics
            this.modelVertices = this.patch.Vertices;

            % set initial state variables of the AGV
            this.position = position;
            this.rotation = rotation;

            this.forwardVector = [1 0 0];
            this.velocity = 1.5;
            this.angularVelocity = 1.2;


            % place patch to its initial position
            this.refreshPatch();


            % initialize tasks
            this.currentTask = {AGVTask.IDLE, 0, 0};
            this.tasklist = Stack();

            % store planner handle (own graph, with the same nodes but
                different adjmat!!)
            this.planner = planner;


            this.workstations = workstations;
        end

    end

    % methods required by the simulation framework ------------------------
    methods

        % Returns the (estimated) time required for the current task to
        % complete. If there is no such time (eg. IDLE), inf is returned.
        % This is used by framework to ensure completition of tasks that
        % require less than the standard simulation timestep.
```

```
function time = timeToFinishTask(this)
    switch this.currentTask{1}
        case AGVTask.GO_STRAIGHT
            time = norm([this.currentTask{2}, this.currentTask{3}])
                / this.velocity;

        case AGVTask.TURN
            time = abs(this.currentTask{2} / this.angularVelocity);

        case AGVTask.WAIT
            time = abs(this.currentTask{2});

        otherwise
            time = inf;
    end
end



% Makes a dt timestep in the simulation
% - executes the AGV's current task
% - if task is completed, pops the next from the stack
% - if the stack is empty, IDLE task is assigned

function [event] = update(this, dt)
    event = [];

    % executes the current task

    taskCompleted = false;
    switch this.currentTask{1}
        case AGVTask.GO_STRAIGHT

            this.forwardVector = [this.currentTask{2}, 0, this.
               currentTask{3}] / norm([this.currentTask{2}, 0,
               this.currentTask{3}]);

            dr = this.velocity * dt * (this.rotation * this.
               forwardVector')';
            this.position = this.position + dr;

            this.refreshPatch();

            this.currentTask{2} = this.currentTask{2} - this.
               velocity * dt * this.forwardVector(1,1);
            this.currentTask{3} = this.currentTask{3} - this.
               velocity * dt * this.forwardVector(1,3);
            taskCompleted = norm([this.currentTask{2}, this.
               currentTask{3}]) < eps;
        case AGVTask.TURN

            angle = sign(this.currentTask{2}) * this.
               angularVelocity * dt;
            this.rotation = this.rotation * angle2dcm(angle, 0, 0);

            this.refreshPatch();
```

```
                this.currentTask{2} = this.currentTask{2} - angle;
                taskCompleted = abs(this.currentTask{2}) < eps;

            case AGVTask.WAIT
                this.currentTask{2} = this.currentTask{2} - dt;
                taskCompleted = abs(this.currentTask{2}) < eps;


            case AGVTask.IDLE
                taskCompleted = false;
        end

        % pops next task if the previous one completed
        if taskCompleted
            task = this.tasklist.pop();
            if ~isempty(task)
                this.currentTask = task;
            else
                this.currentTask = {AGVTask.IDLE, 0, 0};
                event = struct('agv', this, 'message', 'stopped');
            end
        end
    end
end
%-------------------------------------------------------------------------

% methods for AGV manipulation by control system ------------------------
methods

    % Adds task to the AGV's tasklist.
    % The tasks are added to the top of the AGV's task stack in normal
    % order (the first task in the list is executed first). Adding new
    % tasks immediately aborts the AGV's current task, and the
    % execution begins. This requires special attention when adding
    % multiple tasks one after another - instead, use a tasklist!

    function [] = addTasks(this, tasklist)
        if isempty(tasklist)
            return;
        end

        this.tasklist.push(tasklist);
        this.currentTask = this.tasklist.pop();
    end

    % Given 2 adjacent planner graph nodes, returns the time required
    % by this AGV to travel the respective edge

    function [time] = traversalTime(this, source, destination)

        if this.planner.vertices(source, 1) == this.planner.vertices(
            destination, 1) % rotation
            time = abs(this.planner.adjmat(source, destination) / this.
                angularVelocity);
        else
```

```
                time = abs(this.planner.adjmat(source, destination) / this.
                    velocity);
            end
        end

        function [] = setRoute(this, source, destination)
            this.source = source;
            this.destination = destination;
        end


    end
    %--------------------------------------------------------------------

    % methods for graphics settings ------------------------------------

    methods (Access = public)
        function [] = rotateModel(this, angle)
            this.modelVertices = (angle2dcm(angle(1), angle(2), angle(3)) *
                this.modelVertices')';
            this.refreshPatch();
        end

    end

    methods (Access = public)
        function [] = resizeModel(this, ratio)
            this.modelVertices = ratio * this.modelVertices;
            this.refreshPatch();
        end

    end


    methods (Access = public)
        function [] = translateModel(this, offset)
            this.modelVertices = this.modelVertices + repmat(offset, [size(
                this.modelVertices, 1), 1]);
            this.refreshPatch();
        end

    end
    %--------------------------------------------------------------------


    % helper methods for AGV movement and manipulation
    methods (Access = private)
        function [] = refreshPatch(this)
            this.patch.Vertices = (this.rotation * this.modelVertices')' +
                repmat(this.position, [size(this.modelVertices, 1), 1]);
        end

    end


end
```

1.2 Simulation.m

```
classdef Simulation < handle

    properties
        figureHandle % handle of figure for the operations
        agvs % this is a vector of all vehicles in the system
        event % the last event occured during the simulation
        time % global simulaiton time
        videoWriter % optional video writer object to save the simulation
    end

    methods

        % Constructor for the Simulation object
        function [this] = Simulation(figureHandle, videoWriter)
            % initialization of graphics environment, on which the
            % simulation is displayed
            this.figureHandle = this.initializeGraphics(figureHandle);

            % initialization of an empty vector of AGV handles
            this.agvs = AGV.empty(1,0);

            this.event = [];
            this.time = 0;

            % if a video writer argument is passed, it is saved to capture
            % the whole simulation
            if nargin == 2
                this.videoWriter = videoWriter;
            else
                this.videoWriter = [];
            end
        end


        % Being passed an AGV handle, adds the agent to the list of
        % simulated AGVs
        function [] = addAGV(this, agv)
            if isempty(this.agvs)
                this.agvs(1) = agv;
            else
                this.agvs(end+1) = agv;
            end
        end




        function [event] = simulate(this)
            Tsim = 0.04; % speed of simulation 25 FPS

            % at the beginning, there is no event set
            event = [];
```

```
tic
while ishandle(this.figureHandle) && isempty(event)
    t1 = toc;

    % this loop computes the time required for each agent to
    % finish the execution of its current movement primitive
    Treq = zeros(size(this.agvs));
    for k = 1:length(this.agvs)
        Treq(k) = this.agvs(k).timeToFinishTask();
    end

    % the least time computed above or the default dimulation
    % timestep is chosen
    Tstep = min(min(Treq), Tsim);


    % for all agents, their update function is called with the
    % chosen timestep. This computes the next state based on
    % their current state and the task being executed
    for k = 1:length(this.agvs)
        % optionally, an event can be returned
        event = this.agvs(k).update(Tstep);

        % if such returned event exists, control is returned to
        % the caller function (for eg. to let a higher level
        % dispatching system assign tasks to an agent entering
        % idle state). The event is returned as well.
        if ~isempty(event)
            break;
        end
    end

    % to ensure constant framerate, MATLAB is forced to display
    % the current state of all agents in the system
    drawnow;

    % global simulation time is increased with the timestep
    % value
    this.time = this.time + Tstep;

    % if a videoWriter object is present, the current frame is
    % added to the video
    if ~isempty(this.videoWriter)
        this.videoWriter.writeVideo(getframe(this.figureHandle))
            ;
    end

    % sleep time ensuring constant framerate is calculated, and
    % applied
    t2 = toc;
    t = t2 - t1;
    if Tstep - t > 0
        pause(Tstep - t);
    end
end
```

```
        end


        % This fuction initializes the default graphics environment
        % Elements like camera position and target can be changed later
        function [figureHandle] = initializeGraphics(this, figureHandle)
            % if no already created figure is given, the algorithm creates
            % a new one
            if nargin < 2, figureHandle = figure(); end
            figure(figureHandle);

            hold on;

            figureHandle.Position = [100 100 800 600];
            % sets axes and grid visibility
            figureHandle.Children.Visible = 'on';
            % disable clipping of objects not in the visible area
            figureHandle.Children.Clipping = 'off';

            grid on;
            axis(15*[-1 1 -1 1 -1 1]);

            % Camera settings
            camlight('headlight');  % light settings
            camva(90);              % view angle

            campos([0 7 7]);        % camera position
            camtarget([2.5 2.5 1.5]); % camera target

            hold off;

        end


    end



end
```

2 Route planning algorithm

2.1 FGraph.m

```
classdef FGraph < handle

    properties(GetAccess = public, SetAccess = public)
        vertices
        adjmat
    end

    methods

        % Factory graph constructor, that creates a rows x columns x height
        % size 3 dimension grid graph, with nodes equidistantly placed
        function [this] = FGraph(rows, columns, height)

            N = rows * columns * height; % the number of nodes in the grid
            distance = 2;                % distance between the nodes

            % calculation of node coordinated with equidistant spacing
            x = 0:distance:(rows -1)*distance;
            y = 0:distance:(columns-1)*distance;
            z = 0:distance:(height-1)*distance;
            [x,y,z] = meshgrid(x,y,z);
            vertices = [y(:) x(:) z(:)];

            R = rows;
            C = columns;
            H = height;

            % connecting nodes at adjacent grid points
            adjmat=false(N,N);
            for i=1:N
                if i-1>=1 && mod(i-1,C)~=0, adjmat(i,i-1)=true; end;
                if i+1<=N && mod(i,C)~=0, adjmat(i,i+1)=true; end;
                if i-C>=1 && mod(i-C+(C-1), R*C)>=C, adjmat(i,i-C)=true;
                   end;
                if i+C<=N && mod(i+(C-1), R*C)>=C, adjmat(i,i+C)=true; end;
                if i-R*C>=1, adjmat(i, i-R*C) = true; end
                if i+R*C<=N, adjmat(i, i+R*C) = true; end
            end;

            % saving the computed values as state variables
            this.vertices=vertices;
            this.adjmat=adjmat;

        end


        % This function plots the three dimension factory graph to the
        % figure given.

        function plot(this, fig)
```

```
        % if no figure is given, a new one is created
        if nargin==1, fig=figure; end
        figure(fig);

        % Generating xy data for the edges. All edges are plotted in
        % one step as ONE line object. This makes the drawing very
        % efficient.
        [i,j]=find(this.adjmat);
        idx=[i';j';ones(1,length(i))];
        idx=idx(:);

        vxy=this.vertices;
        lxy=vxy(idx,:);
        lxy(3:3:end,:)=NaN;

        line(lxy(:,1),lxy(:,2), lxy(:,3),'Marker','o','MarkerFaceColor'
            ,'black');


        % Generating xy data for all arrowheads. The arrowheads are
        % plotted in one step as ONE patch object. This makes the
        % plotting process very efficient and fast.
        axy = lxy;
        for k=1:3:size(axy,1)
            d=lxy(k+1,:)-lxy(k,:);
            dn=0.2*d/norm(d);
            axy(k,:)=lxy(k,:)+0.8*d+dn;

            rotv = [dn(3)*dn(1) dn(3)*dn(2) -dn(1)^2-dn(2)^2];

            if norm(rotv) < eps
                rotv = 0.05 * [1 1 0]/sqrt(2);
            else
                rotv = 0.05 * rotv / norm(rotv);
            end

            axy(k+1,:)=lxy(k,:)+0.8*d+rotv; %rotation by 90 deg
            axy(k+2,:)=lxy(k,:)+0.8*d-rotv; %rotation by -90 deg


        end
        fac=reshape(1:size(axy,1),3,size(axy,1)/3)';
        patch('Vertices',axy,'Faces',fac,'FaceColor','b','LineStyle','
            none');

        % Labeling of nodes
        for i=1:size(vxy,1)
            text(vxy(i,1)+0.2,vxy(i,2)+0.2, vxy(i,3)+0.2,num2str(i));
        end
    end


    function add_edges(this,edges)
        for k=1:size(edges,1)
            t=edges(k,1);
```

```
            h=edges(k,2);
            this.adjmat(t,h)=true;
        end;
    end


    function delete_edges(this,edges)
        for k=1:size(edges,1)
            t=edges(k,1);
            h=edges(k,2);
            this.adjmat(t,h)=false;
        end;
    end


    function add_vertices(this,xydata)
        this.vertices=[this.vertices; xydata];
        n=size(this.adjmat,1);
        m=size(xydata,1);
        this.adjmat=[this.adjmat false(n,m); false(m,n+m)];

    end

    function delete_vertices(this,v)
        this.vertices(v,:)=[];
        this.adjmat(v,:)=[];
        this.adjmat(:,v)=[];
    end



end

end
```

2.2 PGraph.m

```
classdef PGraph < handle
    % Class for creating and using the planning graph
    properties
        vertices % list of the vertices of the planning graph
        adjmat   % the adjacency matrix
        factory  % the factory matrix, from which the planner was generated
    end
    methods
        function this = PGraph(factory)
            % Generating the vertices. For each factory vertex p, we com-
            % pute all possible orientations that the AGV can take at that
            % vertex. The new vertices are stored in the vertices array.
            % Each row is of length 2 and stores the label [p,a] of the
            % vertex: p is the index of the factory vertex and a is the
            % orientation in radian.
            angle=@(v) atan2(v(2),v(1));
            l=1; vert=[];
            for k=1:size(factory.vertices,1)
                ie=find(factory.adjmat(:,k));
                oe=find(factory.adjmat(k,:));
                a=zeros(1,length(ie)+length(oe));
                j=1;
                for i=1:length(ie)
                    a(j)=angle(factory.vertices(k,:)-factory.vertices(ie(i)
                        ,:));
                    j=j+1;
                end
                for i=1:length(oe)
                    a(j)=angle(factory.vertices(oe(i),:)-factory.vertices(k
                        ,:));
                    j=j+1;
                end
                a=unique(a);
                na=length(a);
                vert(l:l+na-1,:)=[repmat(k,na,1) a'];
                l=l+na;
            end

            nv=size(vert,1);
            adjmat=zeros(nv,nv);

            % Adding edge between every vertex pair belonging to the same
            % factory vertex. These edges represent rotations while the agv
            % is sitting at the vertex point. Technically, an edge is
                created
            % between (p,a) and (q,b) if p==q.
            angle = @(a,b) atan2(-sin(a).*cos(b)+cos(a).*sin(b),cos(a).*cos
                (b)+sin(a)*sin(b));
            for k=1:size(factory.vertices,1)
                l=find(vert(:,1)==k);
                if size(l, 1) == 1
                    continue;
                end
```

```
            J=nchoosek(I,2);
            for i=1:size(J,1)
                adjmat(J(i,1),J(i,2))=angle(vert(J(i,1),2),vert(J(i,2)
                    ,2));
                adjmat(J(i,2),J(i,1))=angle(vert(J(i,2),2),vert(J(i,1)
                    ,2));
            end
        end

        % Adding edges representing translation. Adding edge between
        % vertices (p,a) and (q,b) if a==b and there is an edge between
        % p and q in the factory graph.
        angle=@(v) atan2(v(2),v(1));
        [t,h]=find(factory.adjmat);
        for k=1:length(t)
            tk=factory.vertices(t(k),:);
            hk=factory.vertices(h(k),:);
            diff = hk-tk;
            if norm(diff(1:2)) > eps % if the two edges are not
                vertically aligned
                a=angle(diff);
                i= (vert(:,1)==t(k)) & abs(vert(:,2)-a)<1e-14 ;
                j= (vert(:,1)==h(k)) & abs(vert(:,2)-a)<1e-14 ;
                adjmat(i,j)=norm(hk-tk);
            else % one of the edges is exactly above the another one
                i = find(vert(:,1) == t(k));
                a = vert(i, 2);
                for m = 1:length(i)
                    j = (vert(:,1)==h(k)) & abs(vert(:,2)-a(m))<1e-14;
                    adjmat(i(m), j) = norm(diff);
                end
            end

        end
        this.vertices=vert;
        this.adjmat=adjmat;
        this.factory = factory;
    end


    % Displays the edges of the graph
    % This function is not directly used in the simulation, but is
    % written for testing purposes
    function disp(this)

        [I,J]=find(this.adjmat);
        fprintf('\nTotal number of edges: %2d', length(I));
        fprintf('\n[fv,   ori] ---> [fv,   ori] :  cost \n');
        for k=1:length(I)
            i=I(k); j=J(k);
            if this.vertices(i,1)==this.vertices(j,1)
                if this.adjmat(i,j)>0
                    rotdir='(left)';
                else
                    rotdir='(right)';
                end;
```

```
            else
                if norm(this.factory.vertices(this.vertices(i,1),1:2) -
                    this.factory.vertices(this.vertices(j,1),1:2))<eps
                    rotdir='(vert)';
                else
                    rotdir = '';
                end
            end
            fprintf('[%2d, %+5.2f] --> [%2d, %+5.2f] : %5.2f %s\n',
                this.vertices(i,1),this.vertices(i,2),...
                this.vertices(j,1),this.vertices(j,2),this.adjmat(i,j),
                rotdir);
        end
        fprintf('[fv,   ori] --> [fv,   ori] :  cost \n');
    end

    % Floyd-Warshall algorithm. Output sptable is a cell array,
    % s.t. sptable{i,j} gives the shortest path (list of
    % pgraph vertices) between vertex i and j. The cost of the
    % pathes is stored in matrix dist. This is not used in the routing
    % algorithm, and was implemented for testing purpose only (static
    % route creation)
    function [sptable,dist]=floyd_warshall(this)

        V=length(this.vertices);
        dist=zeros(V,V)+Inf;
        next=zeros(V,V);
        for i=1:V
            for j=1:V
                if i==j, dist(i,j)=0;
                elseif abs(this.adjmat(i,j))>0
                    dist(i,j)=abs(this.adjmat(i,j));
                    next(i,j)=j;
                end;
            end;
        end

        for k=1:V
            for i=1:V
                for j=1:V
                    if dist(i,j)>dist(i,k)+dist(k,j)
                        dist(i,j)=dist(i,k)+dist(k,j);
                        next(i,j)=next(i,k);
                    end
                end
            end
        end

        sptable=cell(V,V);
        for i=1:V
            for j=1:V
                if next(i,j)==0, continue; end;
                sptable{i,j}=i; k=i;
                while k~=j
                    k=next(k,j);
                    sptable{i,j}(end+1)=k;
```

```
                    end;
                end
            end
        end

        % This helper function creates a clone of the PGraph handle object.
        % It is used to create different graphs for different vehicles,
        % based on their size and capabilities
        function [obj] = copy(this)
            obj = PGraph(this.factory);
            obj.vertices = 1*this.vertices;
            obj.adjmat = 1*this.adjmat;
            obj.factory = this.factory;
        end

    end
end
```

2.3 Resources.m

```
classdef Resources < handle
    % RESOURCES
    % Class for storing time windows for planner graph edges
    % Planner graph edges belonging to the same physical part of the
    % factory (the two directions of an edge, or the edges representing
    % rotation in the same node) are associated with a common resource.
    % This can be easily changed if needed, due to the highly flexible
    % structure of the class

    properties
        % timeWindows is a cell array, with one cell for each resource. In
        % a cell, an n*2 matrix is stored, containing the beginning and
        % ending of a time window (assuming there are n time windows for
        % the given resource)
        timeWindows

        % resource_ids is a PlannerNodes*PlannerNodes matrix, containing
        % the associated resource id for every possible edge in the planner
        % graph. Basically, it is a copy of the adjacency matrix, but
        % instead of the path cost, the resource id is stored.
        resource_ids
    end

    methods

        function this = Resources(factory, planner)
            % RESOURCES - class constructor
            % Based on the given factory and planner object, it generates
            % and assigns the resource ids for any edge in the planner
            % graph, based on the rules detailed in the class description.

            % In the alloc_matrix, we generate a resource_id for every
            % physical resource (aka node and edge) in the factory
            alloc_matrix = 1 .* factory.adjmat;
            counter = 1;

            % forst for the nodes
            for m = 1 : size(alloc_matrix, 2)
                alloc_matrix(m, m) = counter;
                counter = counter + 1;
            end

            % now for the edges
            for m = 1 : size(alloc_matrix, 1)
                for n = m+1 : size(alloc_matrix, 2)

                    % if there is a bidirectional edge, we need a common id
                    % for the two directions
                    if alloc_matrix(m, n)~=0
                        alloc_matrix(m, n) = counter;
                    end

                    if alloc_matrix(n, m) ~= 0
```

```
                alloc_matrix(n, m) = counter;
            end

            if alloc_matrix(m, n) ~= 0 || alloc_matrix(n, m)~=0
                counter = counter + 1;
            end
        end
    end

    % now, we assign the id-s to the planner edges
    this.resource_ids = 1 .* planner.adjmat;

    for m = 1 : size(this.resource_ids, 1)
        for n = 1 : size(this.resource_ids, 2)
            this.resource_ids(m, n) = alloc_matrix(planner.vertices
                (m, 1), planner.vertices(n, 1));
        end
    end

    % for every resource, an all-free time window is generated
    this.timeWindows = cell(counter-1, 1);
    for k = 1 : counter - 1
        this.timeWindows{k} = [0, inf];
    end

end

function y = draw(this)
    % DRAW - creates a barplot showing empty and reserved time
    % intervals (green and red respectively). Time interals not
    % displayed (above max. time) are considered green.

    y = zeros(1,1);
    max_so_far = 0;
    for k = 1 : size(this.timeWindows, 1)
        c = 0;
        for m = 1 : size(this.timeWindows{k}, 1)
            if m > 1 & this.timeWindows{k}(m, 1) ~= this.
                timeWindows{k}(m-1, end)
                c = c+1;
                y(k, c) = this.timeWindows{k}(m, 1) - this.
                    timeWindows{k}(m-1, end);
            end
            c = c+1;
            y(k, c) = this.timeWindows{k}(m, 2) - this.timeWindows{
                k}(m, 1);

        end

        if max_so_far < this.timeWindows{k}(end, 1)
            max_so_far = this.timeWindows{k}(end, 1);
        end

        y(k, c) = max_so_far - this.timeWindows{k}(end, 1);

    end
```

```
            b = bar(y, 'stacked', 'r');


            for k = 1 : size(this.timeWindows, 1)
                c = 0;
                for m = 1 : size(this.timeWindows{k}, 1)
                    if m > 1 & this.timeWindows{k}(m, 1) ~= this.
                        timeWindows{k}(m-1, end)
                        c = c+1;
                    end
                    c = c+1;
                    b(c).FaceColor = 'flat';
                    b(c).CData(k,:) = [0 1 0];
                end
            end
        end


    end
end
```

2.4 algo2.m

```
function dynamic_route = algo2(sourceNode, targetNode, releaseTime, planner
    , resources, agv)

% This function implements the route planning algorithm of Stenzel. Being
% given a request R = (sourceNode, targetNode, releaseTime), and the common
% resource handles, it computes the quickest dynamic route respecting time
% windows.


% Parameters:
% sourceNode - The plannerGraph id of the starting node
% targetNode - The plannerGraph id of the destination
% planner - PlannerGraph object containing all nodes and adj matrix
% resources - Resources object containing free time windows for resources
% note: resources are identified by a unique resource_id, not
% equivalent with node or edge ids. For details, see Resources.m

% Return value:
% Dynamic route from source to target node, or empty array if no route
% exists. Dynamic route is an n*3 matrix, every row containing an edge,
% described by (tail, head, theta) parameters. Head and tail are planner
% graph nodes, theta is the earliest time the edge can be entered.


%% Initialization of the algorithm

% in this array will be the route labels returned
% if no route exists, function returns the empty array
dynamic_route = zeros(0, 3);


% The minimum-first priority queue used by the algorithm
H = PriorityQueue(false);

% This double for loop creates a label for all edges (noted by caligraphic
% L in the thesis). For the sake of simplicity, it is nor just an arrow
% L(e), but an arrow of structures, containing L(e), plus the tail and head
% of the respective edge e.
% Moreover, this loop does the initialization step described in lines 3-6
% in the thesis

k = 1;
% for every node-node pait in the planner graph
for m = 1 : size(planner.vertices, 1)
    for n = 1 : size(planner.vertices, 1)
        % if there is an edge between them
        if planner.adjmat(m, n) ~= 0
            % we construct the L(e) priority queue, plus store head-tail
                info
            edge(k).tail = m;
            edge(k).head = n;
            edge(k).labels = PriorityQueue(false);
```

```
            % initialization step:
            % if the tail of an edge is our source node, we add a label to
            % H and to L(e) as well
            if m == sourceNode
                L = Label(k, releaseTime, inf, []);
                H.insert(releaseTime, L);
                edge(k).labels.insert(releaseTime, L);
            end
            k = k + 1;
        end
    end
end

%% Route computation

% while H not empty (there are unexpanded labels)
while H.size() > 0

    % we pop label with minimal completition time from the heap
    [pri, L] = H.pop();

    % this is just for simplicity
    k = L.edgeId;
    tail = edge(k).tail;
    head = edge(k).head;
    rsid = resources.resource_ids(tail, head);

    % if we popped an edge starting from the targetNode, it means, that
    % we have already found the fastest route to it. In this case, the
    % algorithm constructs the path moving backwards along the labels
    % From this function, only the labels are return. Construction of the
    % path, respecting the time windows, is the caller-sides responsibility
    if tail == targetNode

        while isempty(L) == false
            %disp(L);
            dynamic_route(end+1, :) = [edge(L.edgeId).tail, edge(L.edgeId).
                head, L.A];
            L = L.pred;
        end
        dynamic_route = dynamic_route(end:-1:1, :);

        dynamic_route(1, 3) = dynamic_route(2, 3) - agv.traversalTime(
            dynamic_route(1,1), dynamic_route(1,2));%abs(planner.adjmat(
            dynamic_route(2,2), dynamic_route(2,1)));

        break;
    end

    % If we have not reached the goal yet, we expand our label

    % foreach empty time window on the edge examinated
    for t = 1 : size(resources.timeWindows{rsid}, 1)

        % If the last possible arrival is sooner than first time window, we
```

```
% cannot go further on this way. The algorithm moves on to the next
% possible label in H.
if L.B < resources.timeWindows{rsid}(t, 1)
    break;
end

% If the time window closes before the first possible arrival time,
% the algorithm moves on to the next window. Note, windows are
% stored sorted.
if L.A > resources.timeWindows{rsid}(t, 2)
    continue;
end

% The next 3 lines compute first possible time of entering and
% leaving the edge
% First possible enter time: determined by arrival time and start
% of the window
timeIn = max(L.A, resources.timeWindows{rsid}(t, 1));
% First possible leave time: determined by first possible enter
% time + traversal time of the edge. Traversal time is computed
% base on the AVG.
traversalTime = agv.traversalTime(tail, head);
timeOut = timeIn + traversalTime;

% The algorithm checks whether the edge traversal fits into the
% time window. If not, moves to the next possible window.
if timeOut < resources.timeWindows{rsid}(t, 2)

    % The algorithm examines all outgoing edges, and creates a
    % label where necessary
    for r = 1 : size(edge, 2)   % for all edges...
        if edge(r).tail == head  % ...starting from the head of the
            current edge

            % The new label is created
            % First possible arrival to the node: timeOut
            % Last possible arrival time: when the time window
            % closes
            Lnew = Label(r, timeOut, resources.timeWindows{rsid}(t,
                2), L);

            % This is used to determine, whether the new label
            % should be added, or it is inferior to one already
            % added.
            dominated = false;

            valList = edge(r).labels.valueList;
            toBeDeleted = Label.empty(0,1);


            % foreach label of the edge
            for q = 1 : edge(r).labels.numElements;

                Lk = valList{q};

                if isempty(Lk)
```

```
                        continue;
                    end

                    if Lnew.A <= Lk.A && Lk.B <= Lnew.B % <-- Lnew.B
                        toBeDeleted(end+1) = Lk;
                    end

                    % if an already present dominates the new one, no
                    % further checks should be done, we move on to the
                        next
                    % outgoing edge
                    if Lk.A <= Lnew.A && Lnew.B <= Lk.B
                        dominated = true;
                        break;
                    end
                    % end foreach label
                end


                % if the new label was not dominated by an already
                % present one, it is inserted
                if dominated == false

                    % delete the dominated old labels

                    for itt = 1 : size(toBeDeleted, 1)
                        H.erase(toBeDeleted(itt));
                        edge(r).labels.erase(toBeDeleted(itt));
                    end


                    H.insert(Lnew.A, Lnew);
                    edge(r).labels.insert(Lnew.A, Lnew);
                end
                % end of outgoing edges
            end
        end
        % end if we fit in time window
    end
    % end foreach empty windows
end
% end H not empty
end

if isempty(dynamic_route)
    disp('no route to destination');
end

% end of function
end
```

2.5 algo3.m

```
% This is responsible for reserving time windows...
function [] = algo3(dynamic_route, planner, resources)

% for every edge in the dynamic path
for k=1:size(dynamic_route, 1)-1

    v1=planner.vertices(dynamic_route(k, 1)); % tail of edge
    v2=planner.vertices(dynamic_route(k, 2)); % head of edge

    confl = zeros(0, 2);

    % list all conflicting edges, that must be reserved
    for m = 1 : size(planner.vertices, 1)
       for n = 1 : size(planner.vertices, 1)
          if planner.adjmat(m, n) ~= 0
             v3 = planner.vertices(m, 1);
             v4 = planner.vertices(n, 1);
             if v1 == v3 || v1 == v4 || v2 == v3 || v2 == v4
                 confl(end+1, :) = [m, n];
             end
          end
       end
    end

    timeIn = dynamic_route(k, 3);
    timeOut = dynamic_route(k+1, 3);

    % foeach conflicting edge, a reservation is done
    for r = 1 : size(confl, 1)
        rsid = resources.resource_ids(confl(r, 1), confl(r, 2));

        for q = 1 : size(resources.timeWindows{rsid}, 1)
           if timeOut <= resources.timeWindows{rsid}(q, 1)
              continue;
           end

           % when the time window to split is found, the new windows are
           % calculated as described in resource allocation
           if timeIn >= resources.timeWindows{rsid}(q, 1) && timeOut <=
               resources.timeWindows{rsid}(q, 2)
              resources.timeWindows{rsid}(q+1:end+1, :) = resources.
                  timeWindows{rsid}(q:end, :);
              resources.timeWindows{rsid}(q, 2) = timeIn;
              resources.timeWindows{rsid}(q+1, 1) = timeOut;
           end

        end
    end

end
end
```

2.6 dyn_route2task.m

```
function tasklist = dyn_route2task(dynamic_route, releaseTime, planner,
    factory, agv)

% Being given a dynamic route, this function creates a list of movement
% primitives for an agent.


tasklist=cell(10,3);
l=1;

% if the route is empty, no list can be calculated
assert(size(dynamic_route, 1) > 0);

fprintf('ReleaseTime: %d\n', releaseTime);
fprintf('startTime: %d\n', dynamic_route(1,3));

% if the release time is smaller than the first theta value in the route,
% the agent must wait till it can leave the node it resides in
if releaseTime < dynamic_route(1, 3)
    tasklist(l, :) = {AGVTask.WAIT, dynamic_route(1, 3) - releaseTime, 0};
    l = l+1;
end


for k=1:size(dynamic_route, 1)-1

    v1=planner.vertices(dynamic_route(k, 1)); % tail of edge
    v2=planner.vertices(dynamic_route(k, 2)); % head of edge
    a=planner.adjmat(dynamic_route(k,1), dynamic_route(k, 2)); % weight of
        the edge

    % traversal time of the AGV on the edge, depending on its speed
    traversalTime =  agv.traversalTime(dynamic_route(k, 1), dynamic_route(k
        , 2));


    if v1(1)==v2(1) % if two virtual nodes belong to the same phisical one,
        a TURN operation takes place
        if dynamic_route(k+1, 3) > dynamic_route(k, 3) + traversalTime
            tasklist(l, :) = {AGVTask.WAIT, dynamic_route(k+1, 3) -
                dynamic_route(k, 3) - traversalTime, 0};
            l = l+1;
        end

        task={AGVTask.TURN, -a, 0};
        tasklist(l,:)=task;
        l=l+1;

    else %else, for edge traversal, a GO_STRAIGHT is needed


        start = factory.vertices(v1, :);
        stop = factory.vertices(v2, :);
```

```
        ah = norm(stop(1:2) - start(1:2)); % horizontal distance
        av = (stop(3)-start(3)); % vertical distance


        if dynamic_route(k+1, 3) > dynamic_route(k, 3) + traversalTime

            % if the traversal time added to the time of entrance is less
            % than the theta value of the next edge, the agv is obliged to
            % wait in the middle the edge

            % go to the middle
            tasklist(l,:)= {AGVTask.GO_STRAIGHT, ah/2,av/2};
            l=l+1;
            % wait
            tasklist(l, :) = {AGVTask.WAIT, dynamic_route(k+1, 3) -
                dynamic_route(k, 3) - traversalTime,0};
            l = l+1;
            % go to the end
            tasklist(l,:)= {AGVTask.GO_STRAIGHT, ah/2,av/2};
            l=l+1;
        else
            % if no wait is required
            task = {AGVTask.GO_STRAIGHT, ah, av};
            tasklist(l,:)=task;
            l=l+1;
        end
    end

end
tasklist(l:end,:)=[];

end
```

3 Test scripts

3.1 demo_gyor.m

```
%% Initializing environment and graphics
clear
close
clc

addpath(genpath('../framework/'));
addpath(genpath('../util/'));
addpath(genpath('../stenzel/'));


quadPath = 'models/quadcopter_tf.stl';
agvPath = 'models/teljes.stl';

factoryFigure = figure(1);
figure(factoryFigure);

simulation = Simulation(factoryFigure);


cp = [3.0561    -0.1886     2.1911];
ct = [4 4 0.2];

% alternative camera position
% cp = [6 6 6];
% ct = [6 6 0];

campos(cp);
camtarget(ct);
camva(90);

%% Creating the environment

% Create the factory graph
factory = FGraph(1,1,1);
factory.vertices(:, 3) = factory.vertices(:, 3);

% These nodes are added manually, to match the loaded floorplan
factory.add_vertices([2.1 0.8 0]);
factory.add_vertices([2.66 2.66 0]);
factory.add_vertices([3.75 3.75 0]);
factory.add_vertices([5.5 4 0]);
factory.add_vertices([7.5 4 0]);
factory.add_vertices([7.5 5.5 0]);
factory.add_vertices([3 5.25 0]);
factory.add_vertices([5.5 5.5 0]);
factory.add_vertices([5.8 1.9 0]);
factory.add_vertices([5.5 7.5 0]);
factory.add_vertices([7.5 7.5 0]);
factory.add_vertices([5.5 9.5 0]);
factory.add_vertices([7.5 9.5 0]);
```

```
factory.add_vertices([3.5 10.5 0]);
factory.add_vertices([2.25 9.7 0]);
factory.add_vertices([2.25 11.7 0]);
factory.add_vertices([6 11 0]);
factory.add_vertices([9.3 10.8 0]);
factory.add_vertices([9.5 9.4 0]);
factory.add_vertices([9.1 5.3 0]);
factory.delete_vertices(1);


% bidirectional edges between adjacent nodes are added
edges_to_add = [1 2;2 3;3 4;4 5;5 6;3 7;4 8;6 8;7 8;4 9;5 9;8 10;6 11;10
    12;11 13;10 11;12 13;12 14;14 15;14 16;15 16;12 17;13 17;14 17;13 18;18
     19;13 19;11 19;6 20;];
edges_to_add = [edges_to_add; edges_to_add(:, [2 1])];
factory.add_edges(edges_to_add);

% nodes in the air are added sepparately
factory.add_vertices([3.75 3.75 1]);
factory.add_vertices([5.5 4 1]);
factory.add_vertices([7.5 4 1]);
factory.add_vertices([7.5 5.5 1]);
factory.add_vertices([5.5 5.5 1]);
factory.add_vertices([5.5 7.5 1]);
factory.add_vertices([7.5 7.5 1]);
factory.add_vertices([7.5 9.5 1]);
factory.add_vertices([5.5 9.5 1]);
factory.add_vertices([6 11 1]);
factory.add_vertices([3.5 10.5 1]);

% finally, bidirectional edges connecting aerial and ground nodes are added
edges_to_add = [14 31; 17 30; 15 31; 16 31; 30 31;12 29; 29 30; 29 31; 28
    13; 27 11;28 27; 28 30; 28 29;26 10; 26 29; 8 25; 6 24;25 26; 24 27; 3
    21; 25 21];
edges_to_add = [edges_to_add; edges_to_add(:, [2 1])];

factory.add_edges(edges_to_add);
factory.delete_vertices([22 23]);

% the graph is plotted
hold on;
factory.plot(factoryFigure);
hold off;

% Outlook of the factory cell in gyor is loaded (graphical elements)
load('gyar.mat');
ccp = zeros(size(cc) + [2 2]);
ccp(2:end-1, 2:end-1) = cc;
cc = ccp;
[mx, my] = meshgrid(1:size(cc, 2), 1:size(cc,1));
mx = mx / 12;
my = my / 12;
cc = cc / max(max(cc));
colormapR = zeros([size(cc,1), size(cc, 2)]);
colormapG = zeros(size(colormapR));
colormapR(cc>=1) = 1;
```

```
colormapG(cc<=eps) = 1;
colormap = zeros(size(cc,1), size(cc,2), 3);
colormap(:, :, 1) = colormapR;
colormap(:, :, 2) = colormapG;
colormap(:, [1 end], 1) = 1;
colormap(:, [1 end], 2) = 0;
colormap([1 end], :, 1) = 1;
colormap([1 end], :, 2) = 0;

hold on;
s = surf(mx, my, cc, colormap, 'FaceAlpha', 0.5);
s.EdgeColor = 'none';
hold off;

% workstations are drawn
drawWorkstations(factory, [1 9 16 18 20]);


% draws annotation box to show dispatched destinations in the top left
% conrner
hold on;
dim = [0.05 0.65 0.3 0.3];
str = {'', '', '', '', ''};
an = annotation('textbox',dim,'String',str,'FitBoxToText','on', 'FontSize',
    14);
hold off;


%% Initializing Stenzel's algorithm

% The planner graph is created
planner = PGraph(factory);

% The Resources object (containing time windows) is created
resources = Resources(factory, planner);

% Place of the stations is defined
groundStations = [2 91 102 45 110];
airStations = [];
allStations = [airStations, groundStations];

% Planner graph excluding aerial edges is added for ground-bound vehicles
plannerAGV = planner.copy();
plannerAGV.adjmat(factory.vertices(planner.vertices(:,1), 3) > 0, :) = inf;
plannerAGV.adjmat(:, factory.vertices(planner.vertices(:,1), 3) > 0) = inf;

% loading the AGVs to some predefined positions
pposToCoord = @(x)factory.vertices(planner.vertices(x, 1), :);
pposToAngle = @(x)planner.vertices(x, 2);

agv1 = AGV(quadPath, pposToCoord(2), angle2dcm(-pposToAngle(2), 0, 0),
   planner, allStations);
agv1.setRoute(2, 2);
agv1.resizeModel(0.7)
agv1.translateModel([0 0 0.1]);
```

```
agv2 = AGV(quadPath, pposToCoord(91), angle2dcm(-pposToAngle(91),0, 0),
    planner, allStations);
agv2.setRoute(91, 91);
agv2.resizeModel(0.7);
agv2.translateModel([0 0 0.1]);

agv3 = AGV(agvPath, pposToCoord(102), angle2dcm(-pposToAngle(102),0, 0),
    plannerAGV, groundStations);
agv3.rotateModel([pi/2, pi/2, 0]);
agv3.translateModel([0 0 0.5]);
agv3.setRoute(102, 102);

agv4 = AGV(agvPath, pposToCoord(45), angle2dcm(-pposToAngle(45),0, 0),
    plannerAGV, groundStations);
agv4.rotateModel([pi/2, pi/2, 0]);
agv4.translateModel([0 0 0.5]);
agv4.setRoute(45, 45);

reserved = zeros(size(planner.vertices, 1));
reserved([2 91 102 45]) = 1;

% assigning random requests and routes to the AGVs, expect the first one
% task that will be assigned later

releaseTime = 0;
reserved = randomRouteToAGV(agv2, releaseTime, planner, resources, reserved
    , an);
reserved = randomRouteToAGV(agv3, releaseTime, planner, resources, reserved
    , an);
reserved = randomRouteToAGV(agv4, releaseTime, planner, resources, reserved
    , an);

% adding all AGVs to the simulation system
simulation.addAGV(agv1);
simulation.addAGV(agv2);
simulation.addAGV(agv3);
simulation.addAGV(agv4);

% As initialization, an idle AGV (agv1) is chosen
counter = 1;
event = [];
agv = agv1;


%% Main simulation cycle

while ishandle(factoryFigure) && counter < 10

    % new random request is dispatched to the idle AGV
    % the route is calculated, reservations are made
    releaseTime = simulation.time;
    reserved = randomRouteToAGV(agv, releaseTime, planner, resources,
        reserved, an);

    % simulation is resumed until the next agv enters idle state
```

```
    event = simulation.simulate();
    % the idle agv will be the one in the event structure, so next this one
    % is assigned a random request
    agv = event.agv;
end


%% ------------HELPER FUNCTIONS-----------------------------------

function drawWorkstations(factory, vertices)
% This function draws a 0.6 x 0.6 size green square to the map,
% representing worksplaces / parking places


parkplacev = @(x)([x + [-0.6 -0.6 -0.1]; x + [0.6 -0.6 -0.1]; x + [-0.6 0.6
    -0.1]; x + [0.6 0.6 -0.1]]);

for k = 1 : length(vertices)
    vert = parkplacev(factory.vertices(vertices(k), :));
    fac = [1 2 3; 2 3 4];
    patch('Vertices',vert,'Faces',fac,'FaceColor','g', 'LineStyle','none');

end

end


% This function picks a random free parking place, calculates a route to
% the target using the route planning algorithm and makes the requiered
% reservation of resources, so that collision with another vehicles is
% prevented
function [reserved] = randomRouteToAGV(agv, releaseTime, planner, resources
    , reserved, an)
source = agv.destination;
destination = source;

% random destination is generated, until a free parking place is found
while source == destination || reserved(destination)
    destination = agv.workstations(floor(rand()*length(agv.workstations))
        +1);

end

% reservation of parking places is set accordingly
reserved(source) = 0;
reserved(destination) = 1;

% the dynamic route is calculated
dyn_route = algo2(source, destination, releaseTime, planner, resources, agv
    );

% the list of movement primitives is generated from the dynamic route
tasklist = dyn_route2task(dyn_route, releaseTime, planner, planner.factory,
    agv);

% reservations of time windows are made
```

```
algo3(dyn_route, planner, resources);

% the AGV is given the new list of tasks to follow the route
agv.addTasks(tasklist);
agv.setRoute(source, destination);

% the result of the route planning (source->destination) is dispalyed
str = {an.String{2:end}, sprintf('%d, %.2f -> %d, %.2f', planner.vertices(
    source, 1), planner.vertices(source, 2), planner.vertices(destination,
    1), planner.vertices(destination, 2))};
an.String = str;

end
```