Algorithm of Multi-AGVs Flow Optimisation

Emmanuel RIALLAND - ALBA INTELLIGENCE HONG KONG

Contents

```
# Make sure to install RCall, Suppressor
#
#  julia_setup(JULIA_HOME = "/home/emmanuel/Development/_langs/julia/current/bin")
#
JuliaCall::julia_markdown_setup(JULIA_HOME = "/home/emmanuel/Development/_langs/julia/current/bin")
```

```
## Julia version 1.6.0 at location /home/emmanuel/Development/_langs/julia/julia-1.6.0/bin will be used

## Loading setup script for JuliaCall...

## Finish loading setup script for JuliaCall.
```

1 [TODO]

- [TODO: Consider variants to A*]

2 NOTES

- Single planning (next task allocation): matrix 2D because only sorting by distance (cost = time) without considering collisions.
- Multiple planning: 3D
- T_{Max} is currently fixed at 300s. It could just be a minimum to be dynamically increased if A* search is unsuccessful.

```
Tmax = 5 * 60
```

```
## 300
```

3 Assumptions

- Everything in SI.
- Time-step short enough to never have jump-overs even at full speed.
- AGV dimensions approximated in multiple of twice the grid so that all dimensions / tests can be done as centre location +/- value in integers.
- AGV dimensions include palettes.
- Should include security perimeter monitored by AGV?
- All standardised dimensions in floating point. All simulation dimensions in integers. Type mistakes are easier to pick up.
- Dimensions of the floor plan is in units of the gridsize.

4 Data structures

4.1 Data types

4.1.1 Location / Time

$TLocation$:

- $x, y :: Float32$ for standardised units
- $simx, simx :: Int32$ for simulation units (more than enough >= 2,000 km at 1 mm resolution)

```julia
struct TLocation
  x::Float32
  y::Float32
  simx::Int32
  simy::Int32
end
```

$TTime$:

- t :: $Float32$ for standardised units
- $simt$ :: $Int32$ for simulation units (more than enough >= 24 days at 1ms resolution)

```julia
struct TTime
  t::Float32
  simt::Int32
end
```

4.1.2 Direction of an AGV

$TDirection$:: - Enum: [Up, Down, Left, Right]

```julia
@enum TDirection begin
  Up
  Down
  Left
  Right
end
```

4.2 Data structures

4.2.1 Parkings:

- Parking ID
- Parking location: $TLocation$

```julia
struct TParking
  ID::String
  loc::TLocation
end
```

4.2.2 Tasks: Know at algo start

- Task ID

[TODO: Transform to data type]

- Location start (s_j): $TLocation$, $onrack$, $level$. $level$ only in on racks ($onrack == true$)
- Location target (g_j): $TLocation$, $onrack$, $level$. $level$ only in on racks ($onrack == true$)

```
struct TPalette
  loc::TLocation
  onRack::Bool
  level::Int16
end

struct TTask
  ID::String
  start::TPalette
  target::TPalette
end
```

4.2.3 AGV:

- ID number: Known at algo start
- Initial position: $TLocation$ Known at algo start
- Current position: $TLocation$.
- Final position: $TLocation$.
- Final time stamp: $TTime$
- State:
  - *Direction*: $TDirection$
  - *isLoaded*: *Bool*
  - *isSideways*:: *Bool* - is the AGV facing a shelf but otherwise ready to go (about to load or after loading, but no idea where to go).
  - [TODO: More TBD]
- List of tasks as list of tuples allocated to that AGV: Initialised empty
  - Task ID
  - time $TTime$ in simulation units from previous step to reach task. The list is updated at each iteration using the release time calculated by A*. $t = 0$ at the start.
- Allocated parking $Parking_{ID}$.

```
struct TAGV
  ID::String
  start::TLocation
  current::TLocation
  target::TLocation

  time::TTime

  direction::TDirection
  isLoaded::Bool
  isSideways::Bool

  listTasks::Vector{TTask}
  park::TParking
end
```

5 Algorithm

5.1 Expected inputs

All inputs are in Standardised dimensions.

- F: *Required* Floor plan matrix
- $ListAGV$: *Required.* Check list length >= 1
- $ListTasks = []$: *Required.* List length can be 0. Must include the list of tasks that are already allocated at the start of the simulation.
- $ListOfParking = create\_list\_of\_parkings(length(ListAGV))$: *Optional* List of parking slots. Created if missing.
- Speeds: *Required* forward, backward, turn

5.2 Wrapping

First call point is $optimise\_standardised\_units()$.

5.2.1 Timing and Sizing parameters

Estimate adequate time step t_{step} to scaled matrix to M. [CHECK: Default to 200ms]

Max simulation time T_{Max}: This time will be used when planning a path. It has to be long enough so that, given any configuration, A* will find a path from any current position, to achieve any task and go to any parking position. Currently 300s (converted to simulation time using t_{step} in $Step_{Max}$). Track A* failures to check if long enough.

$Step_{Max} = T_{Max}/t_{step}$ is the depth of the 3D planning matrix.

$GridSize$:

- size of each simulation square. AGV dimensions = +/- 1/2 width, +/- 1/2 length.
- Ensures never jumps over even max speed.

5.2.2 Scale everything to Simulation dimensions

Rescale all relevant values from Standardised to Simulation:

- Matrix of the full floor plan: $M = Rescale(F)$
- Same for $ListOfParkings = [Parking_i]$, α_i, s_j, g_j to be appropriately scaled given t_{step}.

```julia
function optimise_standardise_units(F::Matrix{Float32},
                                    listAGVs::Vector{TAGV},
                                    listTasks::Vector{TTask};
                                    AGVLength::Float32 = 3.0,
                                    AGVWidth::Float32 = 2.0,
                                    speedFwd::Float32 = 2.00,
                                    speedBck::Float32 = 1.00,
                                             speedTrn::Float32 = 0.50,
                                             listParkings::Vector{TParking} = create_list_of_parkings(length(lis

    # HARDCODED IN SECONDS. ALL SPEEDS ARE MULTIPLES.
    # THIS WAY: TURN = 1 UNIT. BCK = 2 UNITS. FWD = 4 UNITS.
    TStep = 0.50
    TMax = 300

    # 600 STEPS - MATRIX DEPTH
    StepMax = Int32(ceil(TMax / TStep))

    # GridSize = 50 cm
    GridSize = Int32(ceil(speedTrn / TStep))
    simAGVLength = Int32(ceil(AVGLength / GridSize))
    simAGVWidth = Int32(ceil(AVGWidth / GridSize))

    M = @. Int32(ceil( (1.0/GridSize) * F))

end
```

```
## optimise_standardise_units (generic function with 1 method)
```

5.2.3 Call main algo in simulation units

Call $optimise_simulation_units()$

5.2.4 Scale everything back to standardise units

Rescale all relevant values from Simulation to Standardised

5.2.5 Return

Results are returned in Standardised units.

5.3 $optimise\_\_simulation\_\_units()$ - Preparation of the Main Loop

5.3.1 Preamble

Each AGV will have:

- a list of tasks with a list of release time.

(BUSY WITH TASK — RELEASE TIME)+ — GO TO PARKING

The number of busy times can be 0. Each AGV has a final release time.

5.3.2 Initialisation

- Using M, precompute 2D graph $\mathbb{G}$.
- Using M, precompute M_0 including T_{Max}.
- Initialise matrix τ with dimensions $length(ListTasks) \times length(ListTasks)$ Pre-calculate transfers from the end of one task to the end of another: $\tau_{i,j} = time\_2D(\mathbb{G}, g_i, g_j)$. That matrix is NOT symmetrical.
- For each AGV, if it already has a task, calculate its completion time $time\ 2D(\mathbb{G}, \alpha_i, g_j)$
- Initialise the list of tasks of each AGV to just initial position $PlanningList = [\alpha_i = [\alpha_{i,0}]]$. Each α_i will grow with a list of tasks $\alpha_i = [\alpha_{i,0}, \tau_{i,1}, , \tau_{i,2}, ...]$
- $PlanningList = [\alpha_i]$ will later contain a list of AGVs α_i each containing their allocated tasks (if any). It does not contain the list of parking locations which may vary from iteration to iteration.

5.4 Start Main Loop

$UnAllocatedTasks = ListTasks$ (That is the initial list of tasks)

REPEAT ——————————————————————

5.4.1 Optimal time for the best AGV/Task pair

Planning time starts at time $t :: Int32 = 0$.

5.4.1.1 Assuming some tasks are not allocated yet (we would not be here otherwise. . .) $ListTimePairings = []$ will contain the list of times: time(AGV α_i, unallocated task τ_j)

- For each Task τ_j in $UnAllocatedTasks! = []$ (if $UnAllocatedTasks[]$ is not empty):
  - For each α_i:
    - Take the time of where the AGV is: get the final position $current_i$ of α_i, where $position_i$ is its initial position for the first iteration (if no task), or the position of its final release location g_j given its list of tasks.
    - Take the time to complete the new tasks: $offset_i = time\_2D(\mathbb{G}, \alpha_i, g_i)$ when the list of tasks is empty, $offset_i = \tau_{i,j}$ for i being the last task in α_i.
    - Calculate $time(\alpha_i, g_j) = current_i + offset_i$
    - Push the result (with all relevant information) into $ListTimePairings$

5.4.1.2 Sort $ListTimePairings$ contains all the current α_i with each of them has been allocated a single new task. $ListTimePairings$ may be [].

- Sort all times in $ListTimePairings$ in increasing order and find the shortest. This is the only pair AGV / Task (if any) that will be added to the planning..
- Add the task to that AGV with a time which is the one stored in $ListTimePairings$. Take the previous $PlanningList$ and replace the α_i which has a new task.
- Reorder all the AGVs in decreasing order of total (including updated) release time. Store into $PlanningList$.

  - By construction, that list will contain all the tasks which were previously allocated + only one additional task.
  - The list should only contain entries where there is a new task added to already existing α_i. In other words, choosing an entry will always guarantee that only a single new task is added to the Planning List.
  - We now have a new list of $\alpha_i = [\alpha_{i,0}, \tau_{i,1}, , \tau_{i,2}, ...]$ where AT MOST ONE of them has an additional task. This is the list to be sorted in decreasing order of total time (total release time + time to achieve new task). ONLY ONE if a task was available or NOTHING if the list of tasks was empty to start with.

- Remove the newly allocated task from $UnAllocatedTasks[]$.

5.4.2 Planning Loop

$ListAllocationParking = []$: To store all parking allocation as it happens one by one.

$PlanningListWithParkings$: To store all the α_i with their respective parking before planning.

5.4.2.1 Allocate parking locations Given the list of tasks and the parking position already allocated in $ListAllocationParking$, create list of remaining parking positions accounting.

Looping on the AGVs, allocate to each α_i the closest parking $Parking_i$ from its last $\tau_{i,j}$ in its list of tasks. Push each into $PlanningListWithParkings$.

Sort $PlanningListWithParkings$ in decreasing time order.

5.4.2.2 Start planning loop $ListFullPath = []$: To store all paths one by one.

FOR ————

For each $\alpha_i \in PlanningList$:

- $M_i = M_{i-1}$ (Note that M_0 is precalculated)
- Add slices to M_i so that at least T_{Max} slices of buffer.

Create detailed plan for $Path_i$

- Reset the clock of the AGV: $t_i = 0$
- Plan $PlanningListWithParkings[i]$ on M_i. The planning must record the times of the final realease time at which all tasks for that α_i are completed [CHECK: Is the time of parking to be recorded as well]. The result is $FullPath$. If not solution, raise error to increase T_{Max}.
- Push $FullPath$ into $ListFullPath$
- Push found path $FullPath$ into M_i (i.e. obstruct that path).
- **NEXT**

UNTIL All tasks have been planned

optimise_simulation_units: Return list of $\alpha_i = [\alpha_{i,0}, \tau_{i,1}, , \tau_{i,2}, ...]$, and $ListFullPath$

6 Additional Functions

6.1 $create\_list\_of\_parkings(n)$

Create n parking positions with:

- All values are in standardised dimensions
- Number of parkings located next to sources of tasks in proportion of coming orders. Assume that this being done, equal probability of use is acceptable (no priority in the order of filling the parking).
- All different
- At least one per AGV
- Ensure parking positions do not prevent any traffic once an AGV is occupying it.
  - WARNING: this is necessary to guarantee that searches will always find solution:

6.2 $time\_2D(\mathbb{G}, a, b)$

Time the optimal paths in 2D from location a to b in a precomputed graph $\mathbb{G}$ using A*.

Returns full path and total execution time.

All dimensions are in simulation dimensions.

6.3 $scale\_to\_simulation(object)$

Generic function to rescale the floor plan or object locations from standardised to simulation

6.4 $scale\_to\_standardised(object)$

Generic function to rescale the floor plan or object locations from simulation to standardised