Loosely synchronized rule-based planning

david

June 2024

1 Introduction

Let index set $I = \{1, 2, \ldots, N\}$ denote a set of N agents. All agents move in a workspace represented as a finite graph $G = (V, E)$, where the vertex set V represents all possible locations of agents and the edge set $E \subseteq V \times V$ denotes the set of all the possible actions that can move an agent between a pair of vertices in V. An edge between $u, v \in V$ is denoted as $(u, v) \in E$ and the cost of $e \in E$ is a finite positive real number $cost(e) \in \mathbb{R}^+$. Let $v_o^i, v_d^i \in V$ respectively denote the start and goal location of agent i. Let a superscript $i \in I$ over a variable represent the specific agent that the variable belongs to (e.g. $v^i \in V$ means a vertex with respect to agent i).

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E}) = G \times G \times \cdots \times G$ denote the joint graph which is the Cartesian product of N copies of G, where each vertex $v \in \mathcal{V}$ represents a joint vertex, and $e \in \mathcal{E}$ represents a joint edge that connects a pair of joint vertices. The joint vertices corresponding to the start and goal vertices of all the agents are $v_o = (v_o^1, v_o^2, \cdots, v_o^N)$ and $v_d = (v_d^1, v_d^2, \cdots, v_d^N)$ respectively.

Algorithm 1 Pseudocode for Lsrp

Input: graph G, agents A, starts $\{s_1, \ldots, s_n\}$, goals $\{g_1, \ldots, g_n\}$
Output: paths $\{\pi_1, \ldots, \pi_n\}$
Notation: $State(p, v, t_{from}, t_{to})$
1: $T \leftarrow [0]$; $StateTimeline \leftarrow [\,]$; $Upcoming\pi_{to} \leftarrow \{\}$; $t_{next} \leftarrow next\, t \in T$
2: $StateTimeline_0[i] \leftarrow state(s_i, s_i, 0, 0)$: for each agent $a_i \in A$
3: SET_PRIORITIES(A) ▷ Setting priorities in decreasing order of duration

(for each timestep $t \in T$ until terminates, repeat the following) ▷ T is updated in the end of each loop
4: $\pi_{from} \leftarrow StateTimeline[-1]$; $\pi_{to} \leftarrow$ GET_PI$(Upcoming\pi_{to}, t)$
5: **if** REACH_GOAL(π_{from}) **then return** BACKTRACK(StateTimelist)
6: $p_i \leftarrow$ **if** $\pi_{\text{from}}[i].v = g_i$ **then** ϵ_i **else** $p_i + 1$: for each agent $a_i \in A$
7: $curr\_A \leftarrow$ EXTRACT_AGENT(A, t)
8: sort $curr\_A$ in decreasing order of priorities p_i
9: **for** $a_i \in A$ **do**
10:      **continue: if** $a_i \notin curr\_A$
11:      **if** $\pi_{\text{to}}[i] = \perp$ **then** ASY-PIBT$(a_i, curr\_A, \pi_{from}, \pi_{to}, t, t_{next})$
12: **Update**$(T, StateTimeline, \pi_{to})$

Algorithm 2 Pseudocode for ASY-Pibt

Input: $a_i, curr\_A, \pi_{from}, \pi_{to}, t, t_{next}$

Notation: $Blue\,mark : push\,related.\,Red\,mark : swap\,related$

1: $C \leftarrow \text{Neigh}(\pi_{\text{from}}[i].v) \cup \{\pi_{\text{from}}[i].v\}$
2: sort C in increasing order of $\text{dist}(u, g_i)$ where $u \in C$
3: $a_j \leftarrow \text{SWAP-REQUIRED-POSSIBLE}(a_i, curr_A, \pi_{from}, \pi_{to}, C)$
4: **if** $a_j \neq \bot$ **then** $C.reverse()$
5: **for** $v \in C$ **do**
6:     **if** $\text{OCCUPIED}(\text{v}, \pi_{\text{to}})$ **then continue** $\triangleright$ Check occupation
7:     $a_k \leftarrow \text{PUSH-REQUIRED}(v, curr\_A, \pi_{from}, \pi_{to})$
8:     **if** $a_k \neq \bot$ **then**
9:         $\text{cons\_list} \leftarrow [a_i.\text{curr.v}]$
10:         $t_{\text{wait}}, \pi\text{dict} \leftarrow \text{PUSH-POSSIBLE}(a_k, a_i, curr\_A, \pi_{\text{from}}, \pi_{\text{to}}, \text{cons\_list}, t)$
11:         **if** $t_{wait} = \bot$ **then**
12:             **if** $\pi_{to}[i] \neq \bot$ **then return** $\triangleright$ a_k pulls a_i to swap
13:             **continue**
14:         $\pi_{to}[i] \leftarrow state(\pi_{\text{from}}[i].v, \pi_{\text{from}}[i].v, t, t_{wait})$
15:         $\pi\text{dict}[t_{\text{wait}}]_{\text{to}}[i] \leftarrow \text{state}(\pi_{\text{from}}[i].\text{v}, v, t, t_{\text{wait}} + \text{duration}[a_i])$
16:         **if** $v = C[0] \wedge a_j \neq \bot \wedge \pi_{to}[j] = \bot$ **then**
17:             $t_{move} \leftarrow t_{wait} + duration[a_i]$
18:             $\pi_{to}[j] \leftarrow state(\pi_{\text{from}}[j].v, \pi_{\text{from}}[j].v, t, t_{move})$
19:             $\pi\text{dict}[t_{\text{move}}]_{\text{to}}[j] \leftarrow \text{state}(\pi_{\text{from}}[j].\text{v}, v, t_{\text{move}}, t_{\text{move}} + \text{duration}[a_j])$
20:         $\text{MERGE\_PI}(\pi dict, Upcoming\pi_{to})$
21:         **return**
22:     **if** $v = \pi_{\text{from}}[i].v$ **then**
23:         $\pi_{to}[i] \leftarrow state(\pi_{\text{from}}[i].v, v, t, t_{next})$
24:         **return**
25:     $\pi_{to}[i] \leftarrow state(\pi_{\text{from}}[i].v, v, t, t + duration[a_i])$
26:     **if** $v = C[0] \wedge a_j \neq \bot \wedge \pi_{to}[j] = \bot$ **then**
27:         $t_{move} \leftarrow t + duration[a_i]; \pi_{dict} \leftarrow \{ \}$
28:         $\pi_{to}[j] \leftarrow state(\pi_{\text{from}}[j].v, \pi_{\text{from}}[j].v, t, t_{move})$
29:         $\pi\text{dict}[t_{\text{move}}]_{\text{to}}[j] \leftarrow \text{state}(\pi_{\text{from}}[j].\text{v}, v, t_{\text{move}}, t_{\text{move}} + \text{duration}[a_j])$
30:         $\text{MERGE\_PI}(\pi dict, Upcoming\pi_{to})$
31:     **return**

Algorithm 3 Pseudocodes for Push

1: **procedure** PUSH-REQUIRED($v, curr\_A, \pi_{from}, \pi_{to}$)
2:    **if** $\exists a_k \in curr\_A$ **s.t.** $\pi_{\text{from}}[k].v = v \land \pi_{\text{to}}[k] = \bot$ **then**
3:        **return** a_k
4:    **return** $\bot$

5: Notation : cons_list : vertex occupied by agents in Push $-$ Possible process
6: **procedure** PUSH-POSSIBLE($a_i, a_{pusher}, curr\_A, \pi_{from}, \pi_{to}, cons\_list, t$)
7:    $C \leftarrow \text{Neigh}(\pi_{\text{from}}[i].v)$
8:    sort C in increasing order of $\text{dist}(u, g_i)$ where $u \in C$
9:    $a_j \leftarrow$ SWAP-REQUIRED-POSSIBLE($a_i, curr_A, \pi_{from}, \pi_{to}, C$)
10:    **if** $a_j \neq \bot$ **then** $C.reverse()$ ▷ Swap required
11:    $v_i \leftarrow \pi_{from}[i].v$
12:    **for** $v \in C$ **do**
13:        **if** OCCUPIED(v, π_{to}) **then continue**   ▷ Check occupation
14:        **if** $v \in cons\_list$ **then continue** ▷ Avoid recursion to previous agents
15:        $a_k \leftarrow$ PUSH-REQUIRED($curr\_A, \pi_{from}, \pi_{to}$)
16:        **if** $a_k \neq \bot$ **then**
17:            cons_list $\leftarrow$ cons_list $+ [a_i.curr.v]$
18:            $t_{\text{wait}}, \pi\text{dict} \leftarrow$ PUSH-POSSIBLE($a_k, curr\_A, \pi_{\text{from}}, \pi_{\text{to}}, cons\_list, t$)
19:            **if** $t_{wait} = \bot$ **then continue**
20:            $t_{move\_i} \leftarrow t_{wait} + duration[a_i]$
21:            $\pi_{to}[i] \leftarrow state(v_i, v_i, t, t_{wait})$
22:            $\pi\text{dict}[t_{\text{wait}}]_{\text{to}}[i] \leftarrow state(v_i, v, t_{\text{wait}}, t_{\text{move}\_i})$
23:            **if** $v = C[0] \land a_j \neq \bot \land a_j \neq a_{pusher} \land \pi_{to}[j] = \bot$ **then**
24:                $t_{move\_j} \leftarrow t_{move\_i} + duration[a_j]; v_j \leftarrow \pi_{\text{from}}[j].v$
25:                $\pi_{to}[j] \leftarrow state(v_j, v_j, t, t_{move\_i})$
26:                $\pi dict[t_{move}]_{to}[j] \leftarrow state(v_j, v_i, t_{move\_i}, t_{move\_j})$
27:                **return** $\bot, \bot$ ▷ push not possible
28:            **return** $t_{wait} + duration[a_i], \pi dict$
29:        $t_{move\_i} \leftarrow t + duration[a_i]$
30:        $\pi_{to}[i] \leftarrow state(v_i, v, t, t_{move\_i})$
31:        $\pi_{dict} \leftarrow \{ \}$
32:        **if** $v = C[0] \land a_j \neq \bot \land a_j \neq a_{pusher} \land \pi_{to}[j] = \bot$ **then**
33:            $t_{move\_j} \leftarrow t_{move\_i} + duration[a_j]; v_j \leftarrow \pi_{\text{from}}[j].v$
34:            $\pi_{to}[j] \leftarrow state(v_j, v_j, t, t_{move\_i})$
35:            $\pi dict[t_{move}]_{to}[j] \leftarrow state(v_j, v_i, t_{move\_i}, t_{move\_j})$
36:            **return** $\bot, \bot$ ▷ push not possible
37:        **return** $t + duration[a_i], \pi_{dict}$
38:    **return** $\bot, \bot$ ▷ Push is not possible

4

Algorithm 4 Pseudocodes for Swap

1: **procedure** SWAP-REQUIRED-POSSIBLE($a_i, curr\_A, \pi_{from}, \pi_{to}, C$)
2:     **if** $C[0] = \pi_{from}[i].v$ **then return**$\perp$
3:     $v_i \leftarrow \pi_{from}[i].v$
4:     $a_j \leftarrow$ OCCUPIED($C[0], curr_A, \pi_{from}, \pi_{to}$)
5:     **if** $a_j \neq \perp \wedge \pi_{to}[j] = \perp$ **then**
6:         $v_j \leftarrow \pi_{from}[j].v$
7:         **if** SWAP-REQUIRED(a_i, a_j, v_i, v_j) $\wedge$ SWAP-POSSIBLE(v_j, v_i) **then**
8:             **return** a_j
9:     **for** $u \in Neigh(v_i)$ **do**
10:         $a_k \leftarrow$ OCCUPIED($C[0], curr_A, \pi_{from}, \pi_{to}$)
11:         **if** $a_k = \perp \vee \pi_{from}[k].v = C[0]$ **then continue**
12:         **if** SWAP-REQUIRED($a_k, a_i, v_i, C[0]$) $\wedge$ SWAP-POSSIBLE($C[0], v_i$) **then**
13:             **return** a_k
14:     **return** $\perp$

15: **procedure** SWAP-REQUIRED($a_{push}, a_{pull}, v_{push}, v_{pull}$)
16:     $v_{ps} \leftarrow v_{push}; v_{pl} \leftarrow v_{pull}$
17:     **while** $h(a_{push}, v_{pl}) < h(a_{push}, v_{ps})$ **do**
18:         $n \leftarrow (Neigh(v_{pl})).size()$
19:         **for** $u \in Neigh(v_{pl})$ **do**
20:             $a \leftarrow$ OCCUPIED(u, π_{from}, π_{to})
21:             **if** $u = v_{ps} \vee ((Neigh(u)).size() == 1 \wedge a \neq \perp \wedge a.goal = u)$ **then**
22:                 $n - 1;$**continue**
23:             $next = u$
24:         **if** $n >= 2$ **then return false** $\triangleright$ Push can solve this case
25:         **if** $n <= 0$ **then break** $\triangleright$ Dead end
26:         $v_{ps} \leftarrow v_{pl}; v_{pl} \leftarrow next$
27:     $condition_1 \leftarrow (h(a_{pull}, v_{ps}) < h(a_{pull}, v_{pl}))$
28:     $condition_2 \leftarrow (h(a_{push}, v_{ps}) = 0) \vee (h(a_{push}, v_{pl}) < h(a_{push}, v_{ps}))$
29:     **return** $condition_1 \wedge condition_2$

30: **procedure** SWAP-POSSIBLE(v_{push}, v_{pull})
31:     $v_{ps} \leftarrow v_{push}; v_{pl} \leftarrow v_{pull}$
32:     **while** $v_{pl} \neq v_{push}$ **do** $\triangleright$ Avoid loop
33:         $n \leftarrow (Neigh(v_{pl})).size()$
34:         **for** $u \in Neigh(v_{pl})$ **do**
35:             $a \leftarrow$ OCCUPIED(u, π_{from}, π_{to})
36:             **if** $u = v_{ps} \vee ((Neigh(u)).size() == 1 \wedge a \neq \perp \wedge a.goal = u)$ **then**
37:                 $n - 1;$**continue**
38:             $next = u$
39:         **if** $n >= 2$ **then return true**
40:         **if** $n <= 0$ **then return false**
41:         $v_{ps} \leftarrow v_{pl}; v_{pl} \leftarrow next$
42:     **return false**