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 $\{\mathrm{s}_1, \ldots, \mathrm{s}_\mathrm{n}\}$, goals $\{\mathrm{g}_1, \ldots, \mathrm{g}_\mathrm{n}\}$
Output: paths $\{\pi_1, \ldots, \pi_\mathrm{n}\}$
Notation: $State(p, v, t_{from}, t_{to})$

$T \leftarrow [0]$; $StateTimeline \leftarrow [\ ]$; $Upcoming\pi_{to} \leftarrow \{\}$; $t_{next} \leftarrow next\, t \in T$
$StateTimeline_0[i] \leftarrow state(s_i, s_i, 0, 0)$: for each agent $a_i \in A$
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
$\pi_{from} \leftarrow StateTimeline[-1]$; $\pi_{to} \leftarrow$ GET_PI($Upcoming\pi_{to}, t$)
if REACH_GOAL(π_{from}) **then return** BACKTRACK(StateTimelist)
$p_i \leftarrow$ **if** $\pi_{\mathrm{from}}[i].v = g_i$ **then** ϵ_i **else** $p_i + 1$: for each agent $a_i \in A$
$curr\_A \leftarrow$ EXTRACT_AGENT(A, t)
sort $curr\_A$ in decreasing order of priorities p_i
for $a_i \in A$ **do**
  continue: **if** $a_i \notin curr\_A$
  if $\pi_{\mathrm{to}}[i] = \perp$ **then** ASY-PIBT($a_i, curr\_A, \pi_{from}, \pi_{to}, t, t_{next}$)
Update($T, StateTimeline, \pi_{to}$)

Algorithm 2 Pseudocode for ASY-Pibt

Input: $a_i, curr\_A, \pi_{from}, \pi_{to}, t, t_{next}$
Notation: $Blue\,mark$: $push\,related.$ $Red\,mark$: $swap\,related$

$C \leftarrow \text{Neigh}(\pi_{\text{from}}[i].v) \cup \{\pi_{\text{from}}[i].v\}$
sort C in increasing order of $\text{dist}(u, g_i)$ where $u \in C$
$a_j \leftarrow$ Swap-Required-possible$(a_i, curr_A, \pi_{from}, \pi_{to}, C)$
if $a_j \neq \bot$ **then** $C.reverse()$
for $v \in C$ **do**
    if Occupied$(\text{v}, \pi_{\text{to}})$ **then continue** ▷ Check occupation
    $a_k \leftarrow$ Push-Required$(v, curr_A, \pi_{from}, \pi_{to})$
    if $a_k \neq \bot$ **then**
        $\text{cons\_list} \leftarrow [\text{a}_\text{i}.\text{curr.v}]$
        $t_{wait}, \pi dict \leftarrow$ Push-Possible$(a_k, a_i, curr_A, \pi_{from}, \pi_{to}, cons_list, t)$
        if $t_{wait} = \bot$ **then continue**
        $\pi_{to}[i] \leftarrow state(\pi_{\text{from}}[i].v, \pi_{\text{from}}[i].v, t, t_{wait})$
        $\pi\text{dict}[\text{t}_\text{wait}]_\text{to}[\text{i}] \leftarrow \text{state}(\pi_\text{from}[\text{i}].\text{v}, \text{v}, \text{t}, \text{t}_\text{wait} + \text{duration}[\text{a}_\text{i}])$
        if $v = C[0] \wedge a_j \neq \bot \wedge \pi_{to}[j] = \bot$ **then**
            $t_{move} \leftarrow t_{wait} + duration[a_i]$
            $\pi_{to}[j] \leftarrow state(\pi_{\text{from}}[j].v, \pi_{\text{from}}[j].v, t, t_{move})$
            $\pi\text{dict}[\text{t}_\text{move}]_\text{to}[\text{j}] \leftarrow \text{state}(\pi_\text{from}[\text{j}].\text{v}, \text{v}, \text{t}_\text{move}, \text{t}_\text{move} + \text{duration}[\text{a}_\text{j}])$
        Merge_pi$(\pi dict, Upcoming\pi_{to})$
        return
    if $v = \pi_{\text{from}}[i].v$ **then**
        $\pi_{to}[i] \leftarrow state(\pi_{\text{from}}[i].v, v, t, t_{next})$
        return
    $\pi_{to}[i] \leftarrow state(\pi_{\text{from}}[i].v, v, t, t + duration[a_i])$
    if $v = C[0] \wedge a_j \neq \bot \wedge \pi_{to}[j] = \bot$ **then**
        $t_{move} \leftarrow t + duration[a_i]; \pi_{dict} \leftarrow \{\ \}$
        $\pi_{to}[j] \leftarrow state(\pi_{\text{from}}[j].v, \pi_{\text{from}}[j].v, t, t_{move})$
        $\pi\text{dict}[\text{t}_\text{move}]_\text{to}[\text{j}] \leftarrow \text{state}(\pi_\text{from}[\text{j}].\text{v}, \text{v}, \text{t}_\text{move}, \text{t}_\text{move} + \text{duration}[\text{a}_\text{j}])$
        Merge_pi$(\pi dict, Upcoming\pi_{to})$
    return

Algorithm 3 Pseudocodes for Push

procedure PUSH-REQUIRED($v, curr\_A, \pi_{from}, \pi_{to}$)
  if $\exists a_k \in curr\_A$ **s.t.** $\pi_{\mathrm{from}}[k].v = v \land \pi_{\mathrm{to}}[k] = \perp$ **then**
    return a_k
  return $\perp$

Notation : cons_list : vertex occupied by agents in Push − Possible process
procedure PUSH-POSSIBLE($a_i, a_{pusher}, curr\_A, \pi_{from}, \pi_{to}, cons\_list, t$)
  $C \leftarrow \mathrm{Neigh}(\pi_{\mathrm{from}}[i].v)$
  sort C in increasing order of $\mathrm{dist}(u, g_i)$ where $u \in C$
  $a_j \leftarrow$ SWAP-REQUIRED-POSSIBLE($a_i, curr_A, \pi_{from}, \pi_{to}, C$)
  if $a_j \neq \perp$ **then** $C.reverse()$ ▷ Swap required
  for $v \in C$ **do**
    if OCCUPIED($\mathrm{v}, \pi_{\mathrm{to}}$) **then continue** ▷ Check occupation
    if $\mathrm{v} \in \mathrm{cons\_list}$ **then continue** ▷ Avoid recursion to previous agents
    $a_k \leftarrow$ PUSH-REQUIRED($curr\_A, \pi_{from}, \pi_{to}$)
    if $a_k \neq \perp$ **then**
      $\mathrm{cons\_list} \leftarrow \mathrm{cons\_list} + [\mathrm{a_i.curr.v}]$
      $\mathrm{t_{wait}}, \pi\mathrm{dict} \leftarrow$ PUSH-POSSIBLE($\mathrm{a_k}, \mathrm{curr\_A}, \pi_{\mathrm{from}}, \pi_{\mathrm{to}}, \mathrm{cons\_list}, \mathrm{t}$)
      if $t_{wait} = \perp$ **then continue**
      $t_{move} \leftarrow t_{wait} + duration[a_i]$; $v_i \leftarrow \pi_{from}[i].v$
      $\pi_{to}[i] \leftarrow state(v_i, v_i, t, t_{wait})$
      $\pi\mathrm{dict}[\mathrm{t_{wait}}]_{\mathrm{to}}[\mathrm{i}] \leftarrow \mathrm{state}(\mathrm{v_i}, \mathrm{v}, \mathrm{t_{wait}}, \mathrm{t_{move}})$
      return $t_{move}, \pi dict$
    $t_{move} \leftarrow t_{wait} + duration[a_i]$; $v_i \leftarrow \pi_{from}[i].v$
    $\pi_{to}[i] \leftarrow state(v_i, v, t, t_{move})$
    $\pi_{dict} \leftarrow \{\ \}$
    return t_{move}, π_{dict}
  return $\perp, \perp$ ▷ Push is not possible

Algorithm 4 Pseudocodes for Swap

procedure Swap-Required-Possible($a_i, curr\_A, \pi_{from}, \pi_{to}, C$)
    if $C[0] = \pi_{from}[i].v$ **then return** $\perp$
    $v_i \leftarrow \pi_{from}[i].v$
    $a_j \leftarrow$ Occupied($C[0], curr_A, \pi_{from}, \pi_{to}$)
    if $a_j \neq \perp \wedge \pi_{to}[j] = \perp$ **then**
        $v_j \leftarrow \pi_{from}[j].v$
        if Swap-Required(a_i, a_j, v_i, v_j) $\wedge$ Swap-Possible(v_j, v_i) **then**
            return a_j
    for $u \in Neigh(v_i)$ **do**
        $a_k \leftarrow$ Occupied($C[0], curr_A, \pi_{from}, \pi_{to}$)
        if $a_k = \perp \vee \pi_{from}[k].v = C[0]$ **then continue**
        if Swap-Required($a_k, a_i, v_i, C[0]$) $\wedge$ Swap-Possible($C[0], v_i$) **then**
            return a_k
    return $\perp$

procedure Swap-Required($a_{push}, a_{pull}, v_{push}, v_{pull}$)
    $v_{ps} \leftarrow v_{push}; v_{pl} \leftarrow v_{pull}$
    while $h(a_{push}, v_{pl}) < h(a_{push}, v_{ps})$ **do**
        $n \leftarrow (Neigh(v_{pl})).size()$
        for $u \in Neigh(v_{pl})$ **do**
            $a \leftarrow$ Occupied(u, π_{from}, π_{to})
            if $u = v_{ps} \vee ((Neigh(u)).size() == 1 \wedge a \neq \perp \wedge a.goal = u)$ **then**
                $n - 1;$ **continue**
            $next = u$
        if $n >= 2$ **then return false** ▷ Push can solve this case
        if $n <= 0$ **then break** ▷ Dead end
        $v_{ps} \leftarrow v_{pl}; v_{pl} \leftarrow next$
    $condition_1 \leftarrow (h(a_{pull}, v_{ps}) < h(a_{pull}, v_{pl}))$
    $condition_2 \leftarrow (h(a_{push}, v_{ps}) = 0) \vee (h(a_{push}, v_{pl}) < h(a_{push}, v_{ps}))$
    return $condition_1 \wedge condition_2$

procedure Swap-Possible(v_{push}, v_{pull})
    $v_{ps} \leftarrow v_{push}; v_{pl} \leftarrow v_{pull}$
    while $v_{pl} \neq v_{push}$ **do** ▷ Avoid loop
        $n \leftarrow (Neigh(v_{pl})).size()$
        for $u \in Neigh(v_{pl})$ **do**
            $a \leftarrow$ Occupied(u, π_{from}, π_{to})
            if $u = v_{ps} \vee ((Neigh(u)).size() == 1 \wedge a \neq \perp \wedge a.goal = u)$ **then**
                $n - 1;$ **continue**
            $next = u$
        if $n >= 2$ **then return true**
        if $n <= 0$ **then return false**
        $v_{ps} \leftarrow v_{pl}; v_{pl} \leftarrow next$
    return false