MAPF-LNS2: Fast Repairing for MAPF via Large Neighborhood Search

- Starting with set of paths with collisions, MAPF-LNS2 repeatedly selects a few of agents and replans accordingly to get lesser number of collision until collision-free path.

1. Introduction

- MAPF plans collision-free paths for multiple agent, is a NP-Hard problem to solve optimally and minimizes the travel time of agents.
- Main agendas are :-
  (i) First, an efficient SAPF-algo SIPPS based on SIPP which avoids collision & min. collisions with other paths. Runs 5x the Space-Time A*
  (ii) Propose sub-optimal MAPF-LNS2 which is fast, scalable & memory efficient.

---

2. Background

- MAPF : for a $G = (V, E)$, n agents $A = \{a_1, \ldots, a_n\}$ and start vert a $s_i \in V$, target $g_i \in V$ for each $a_i \in A$. There is always a unit move or wait instruction.
- And we create plans of paths $\{p_1, \ldots, p_n\}$, making feasible plan with small $\sum_{i=1}^{n} |p_i|$.

---

3. MAPF-LNS2

- Large Neighborhood Search (LNS) destroys part of solution, fixes the remaining part. Then repairs and replaces the broken part if sol^n is better and repitition.
- MAPF-LNS improves the sol^n, while MAPF-LNS2 finds the solution, for a MAPF instance.
  (i) MAPF-LNS2 first finds partial / complete plan from MAPF algo.
  (ii) Then, plans path for remaining ones, min-imizing collision with earlier paths.
  (iii) Repeats this process, until feasible plan
- At each iteration, $A_s \subseteq A$ are selected with path P^-, then calls modified MAPF for them & collision with P / P^-.

- Anvil MAPF-LNS2 uses modified Prioritized Planning (PP)
- PP assigns random priorities & replans path one by one and the new paths are denoted as P^+ and replaces $P \rightarrow (P \setminus P^-) \cup P^+$ iff efficient.

4. Pathfinding with Dynamic Objects (PMDO)

- (v,t), (e,t) and $(v,[t,\infty)$ are vertex, edge & target $v \in V$, $e \in E$ & vertex occupied $(t-1)$ to t timestep
- Graph $G=(V,E)$, start vertex $(s \in V)$, target $g \in V$ and two finite sets of Obstacles O^h (hard) and O^s (soft obstacle)
- To find $p: s \rightarrow g$, with no hard collision.
- Hence we minimize soft obstacles.

4.1 Space-Time A* (for PMDO)

- Performs A* search, agent can move from state (v,t) to (v',t'), iff $((v,v') \in E \vee v=v') \wedge t'=t+1$ and no collision with O^h.
- Along with g, h & f values, each node stores c-value, the no. of collision with obs in O^s
- For optim., we sort open list in ascending order of this c-value (f-value tiebreaker)
- For dealing with infinite time, we make space time A* to generate timesteps ~~with~~ no greater than maximum of timesteps of obstacle

$$\max\{t \in \mathbb{N} \mid (v,t) \in O^h \cup O^s \vee (e,t) \in O^h \cup O^s \vee (v,[t,\infty)) \in O^h \cup O^s\}$$

and switch to normal A* with no time,

4.2 SIPPS

- Safe Interval Path Planning is fast form of Space-time A* using time intervals.
- Each state is defined by a vertex & free-time interval. i.e. v and $[a,b)$ and chooses the earliest path (partial) that arrive at v in $[a,b)$ without losing optimality.
- Safe Interval Path Planning with Soft constraints (SIPPS) to solve PMDO.

• Safe intervals
- no hard vertex
- either (soft or target obstacle) or (no soft and no soft target)
- Safe interval table T with seqn of safe intervals T[v], by dividing $[0, \infty)$ in min^m set of disjoint safe intervals in T[v] in chronology.

• SIPPS Nodes:
- n.v (vertex), (n.low, n.high) (safe interval), index n.id (safe interval ⊆ of id^th one) & bool n.is-goal
- f-value = g-value + h-value
  (n.low) (min. time from n.v to g)
- c-value = no. of soft collision from root to node

$$c(n) = c(n') + c_v + c_e$$

where,

$c_v \rightarrow \{1$ if soft obstacle, else $0\}$

$c_e \rightarrow \{1$ if $((n'.v, n.v), n.low) \in O^s$, else $0\}$

Note :- If n is root node, then $c(n) = c_v$

• Heuristic
- of a non-goal node n is given as :-

$$h(n) = \begin{cases} \max\{d(n.v, g), T' - g(n)\}, & c(n) = 0 \\ \max\{d(n.v, g), T - g(n)\}, & c(n) = 1 \end{cases}$$

- h(goal node) = 0

Algorithm 1: SIPPS

```
1  T ← buildSafeIntervalTable(V, O^h, O^s);
2  root ← Node(s, T[s][1], 1, false);      // 1 and false
     indicate that root.id = 1 and root.is_goal = false
3  T ← 0;                                  // Lower bound on travel time
4  if ∃t : (g, t) ∈ O^h then
       T ← max{t | (g, t) ∈ O^h} + 1;
5  compute g-, h-, f-, and c-values of root;
6  Q ← {root}; P ← ∅;      // Initialize open and closed
     lists
7  while Q is not empty do
8      n ← Q.pop();   // Node with the smallest c-value
9      if n.is_goal then return extractPath(n);
10     if n.v = g ∧ n.low ≥ T then
11         c_future ← |{(g, t) ∈ O^s | t > n.low}|;
12         if c_future = 0 then return extractPath(n);
13         n' ← a copy of n with is_goal set to true;
14         c(n') ← c(n') + c_future;
15         INSERTNODE(n', Q, P);          // Algorithm 3
16     EXPANDNODE(n, Q, P, T);            // Algorithm 2
17     P ← P ∪ {n};
18 return "No Solution";
```

with their timesteps being the earliest arrival time of

Algorithm 2: EXPANDNODE$(n, Q, P, \mathcal{T})$

```
1  I ← ∅;
2  foreach v : (n.v, v) ∈ E do
3      I ← I ∪ {(v, id) |
         T[v][id] ∩ [n.low + 1, n.high + 1) ≠ ∅, id ∈ ℕ};
4  if ∃id : T[n.v][id].low = n.high then
5      I ← I ∪ {(n.v, id)};      // Indicates wait actions
6  foreach (v, id) ∈ I do
7      [low, high) ← T[v][id];
8      low ← earliest arrival time at v within
         [low, high) without colliding with edge
         obstacles in O^h;
9      if low does not exist then continue;
10     low' ← earliest arrival time at v within
         [low, high) without colliding with edge
         obstacles in O^h ∪ O^s;
11     if low' exists ∧ low' > low then
12         n_1 ← Node(v, [low, low'), id, false);
13         INSERTNODE(n_1, Q, P) ;          // Algorithm 3
14         n_2 ← Node(v, [low', high), id, false);
15         INSERTNODE(n_2, Q, P);           // Algorithm 3
16     else
17         n_3 ← Node(v, [low, high), id, false);
18         INSERTNODE(n_3, Q, P);           // Algorithm 3
```

Algorithm 3: INSERTNODE(n, Q, P)

```
1  compute g-, h-, f-, and c-values of n;
2  N ← {q ∈ Q ∪ P | q ∼ n};   // Nodes identical to n
3  foreach q ∈ N do
4      if q.low ≤ n.low ∧ c(q) ≤ c(n) then      // q ⪰ n
5          return;                              // No need to generate n
6      else if n.low ≤ q.low ∧ c(n) ≤ c(q) then // n ⪰ q
7          delete q from Q and P;               // Prune q
8      else if n.low < q.high ∧ q.low < n.high then
9          if n.low < q.low then n.high = q.low;
10         else q.high = n.low;
11 insert n into Q;
```

Theorem - 1 SIPPS guarantees to return a path if one exists and "No Solution" otherwise.

Theorem - 2 SIPPS guarantees to return a shortest path with ZERO soft collisions if one exists.

5. Neighborhood Selection

- Current plan (P), neighborhood (A_s), size (A_s) = N
- $G_c = (V_c, E_c)$ is collision graph

where,

$$V_c = \{ i \mid a_i \in A \}$$

$$E_c = \{ (i,j) \mid P_i \in P \text{ collides with } P_j \in P \}$$

- Collision-based nbd
  - We select random $v \in V_c$ with $deg(v) > 0$, find largest connected component $G'_c = (V'_c, E'_c)$ of G_c that contains v.

  Cases:-

  (i) $|V'_c| \leq N$: Add all a_v where $v \in V'_c$ in A_s and add the agents which might collide (repeatedly) until $|A_s| = N$.

  (ii) Otherwise, select random walks of N vertices.

- Failure-based nbd
  - Scenarios for failure

  (i) a_i blocked by other agents 'sitting' on target & blocking all paths.

  (ii) a_i is "run over" by other agents in early timesteps.

  - We first select $a_i \in A$ with probab $\propto deg(i)$.
  - Then we collect two sets of agent

    $\{A^s = \{ a_j \in A \mid P_j \in P \text{ visits } s_i \}\}$

    $\{ A^g = \{ a_j \in A \mid P \text{ visits } g_j \} \}$ $P \to$ path (min)

  - Now if :-

    (i) $|A^s \cup A^g| = 0$, we will get path.

    (ii) $|A^s \cup A^g| < (N-1)$, we add agents repeatedly till $|A_s| = N$ (whose target vertices are in $P_j \in P$)

    (iii) else, we add all in A^g & first $N - 1 - |A^g|$ in A^s.

- Random Nbd
  - Randomly select N agents a_i; ~~priority $\propto deg(a_i)$~~.

    priority $\propto (deg(a_i) + 1)$

  - So, agents who don't collide are also selected

- Adaptive LNS (ALNS)
  - we keep weight w_i for each Nbd selection
  - Initially all $w_i = 1$, then with each iter,
  - $\text{prob}(\text{method } i) = \dfrac{w_i}{\sum_j w_j}$
  - to generate nbd & plan path. After replanning
    $w_i = \gamma \cdot \max\{0, c^- - c^+\} + (1-\gamma) \cdot w_i$
    where, $c^+ \rightarrow$ CP after plan after
    $c^- \rightarrow$ CP before
    $\gamma \in [0,1]$

---

6. Experiments
- Compared with bounded sub-optimal EECBS, PP, PP with random starts & rule-based PPS.
- EECBS* uses SIPPS instead of space-time A*
- MAPF-LNS2 uses :-
  (i) PP for initial plans
  (ii) ALNS of 4 nbd (N=8).
  (iii) SIPPS to plan single agent paths.

- Exp-1
  - MAPF-LNS2 (SIPPS) >> MAPF_LNS2 (space-time A*)
- Exp-2
  - MAPF-LNS-2 with ALNS vs Random vs Failure vs Collision >>
- Exp-3
  - MAPF-LNS2 PP vs PPR (Random) vs LNS <<
- Exp-4
  - PPS vs EECBS vs EECBS* vs PP^R vs MAPF-LNS2 <<
- Exp-5
  - MAPF-LNS2 effects of longer warehouse run-time, & is the best.

7. Summary

- A sub-optimal MAPF-LNS2, which solves by recursively repairing colliding paths, while keeping the other paths fixed.
- MAPF outperforms various state of the art MAPF algo.
- SIPPS MAPF-LNS2 outperforms shaw-him A* MAP-LNS2

(IMT2020133 - Darpan Singh)