CMPT417 Intelligent System

MAPF-LNS2

Team Swarm R&D

Minjae Shin 301383936
Brandon Zhang 301183305
Kukjin Kim 301439624

Problem Introduction

Multi-Agent Pathfinding (MAPF) is the problem of finding the shortest collision-free paths for all simultaneously moving agents in a space. The space is a 2-dimensional grid of cells, and agents have a starting location and a goal location within the coordinates of the space, (x_s, y_s) and (x_g, y_g) respectively. A path is a list where the index is the time and the element is the location of the agent at that time (path[t] = (x_t, y_t)). A collision occurs when either two agents arrive at the same location at the same time, called a vertex collision (path1[t] == path2[t]), or when they both move between the same two locations in the opposite direction at the same time, called an edge collision (path1[t-1] == path2[t] && path1[t] == path2[t-1]). An obstacle is a location that an agent cannot be at a certain time (path[t] != (x_o, y_o)). The cost of a path is the time it takes for the agent to reach their goal location from their start location, and the cost of a solution is the sum of path costs for all agents.

Within this problem definition, solutions can be interested in either total path cost or computation speed, as they commonly trade-off with each other. In this report we implement an algorithm for the suboptimal variant called Large Neighbourhood Search 2 (LNS2)(Li et al. 2022). The suboptimal MAPF problem is interesting for its practical utility as it can be very computationally intensive to solve for optimal cost. Frequently, a solution that is only slightly suboptimal can be found an order of magnitude or more faster, which is perfectly acceptable for many real-world applications, making this a particularly lucrative research subject.

Algorithm

Overview

In this report we implemented the suboptimal MAPF solver LNS2. The algorithm starts by using a greedy pathfinding method to get a plan that can contain collisions. That initial plan goes through iterations of replanning by selecting a subset of agents of user-defined size N, called a neighbourhood, and finds new paths for them using a pathfinding method that attempts to avoid as many collisions as possible. If the new plan has fewer collisions it is kept for future replanning iterations, and returned when it has zero collisions.

Initial Path Plan

We obtained an initial plan by using Prioritised Planning with SIPPS on the neighbourhood of all agents. This allowed the initial plan to include unavoidable collisions while doing some work to minimise their number.

Adaptive Large Neighbourhood Search (ALNS)

To select a subset of agents, the paper selects one of three different selection strategies at each iteration: collision based neighbourhoods, failure based neighbourhoods, and random neighbourhoods. The chance to select a strategy is weighted by a probability that is updated after each iteration proportional to the number of collisions it managed to reduce. newWeight = r * max(0, oldPlanCollisions - newPlanCollisions) + (1-r). r is a hyperparameter that we set to be 0.1.

Collision Based Neighbourhood

We want to find an agent with collisions and replan for it and other agents that collided with it or that are close to it. Create a collision graph and select a random agent and get its connected component. Fill the neighbourhood with agents in the connected component up to the selected neighbourhood size N using a random walk on the connected component. If there is still room in the neighbourhood, select a random position from all paths of agents in the connected component and select the agent closest to that position that is not in the current neighbourhood set, and repeat until the neighbourhood is filled.

Failure Based Neighbourhood

Select a random agent with probability proportional to its number of collisions with other agents + 1. Using this agent as agent 1 we want to find two sets of other agents: A_g is composed of agents that have their goal location on the path of agent 1. A_s is composed of agents that visit agent 1's starting point in their path. In filling the neighbourhood we prioritise agent 1, then A_g agents, then A_s agents due to the A_g condition being more common. If the neighbourhood is not full, iteratively fill it with a random selected agent outside the neighbourhood that has its path overlap with a goal location of the neighbourhood agents.

Random Neighbourhood

Iteratively fill the neighbourhood with agents with probability proportional to its number of collisions +1.

Neighbourhood Pathfinding

The selected neighbourhood defines a subproblem to solve using a complete MAPF algorithm. We implemented the randomly ordered prioritised planning used in the paper as well as adapting the basic Conflict-Based-Search algorithm developed for the individual project for use in this.

Modified Prioritised Planning (PP)

Agents in a neighbourhood are ordered randomly for prioritised planning. Paths of agents outside the neighbourhood were considered as hard obstacles, and paths solved for agents inside the neighbourhood during prioritised planning comprised soft obstacles for subsequent agents.

Safe Interval Path Planning (SIPP)

The authors use a modified Safe Interval Path Planning (SIPP) algorithm that uses soft obstacles for the low-level pathfinding. SIPP (Phillips et al. 2011) is a modified A* search algorithm that uses time intervals instead of the fully expanded space-time graph. Earlier arrival times within an interval at a position dominates latter times so fewer nodes need to be expanded overall. SIPP with soft obstacles is called SIPPS. Soft obstacles are collisions that can be allowed in the final path. There are different MAPF problem formulations that use soft obstacles for different purposes, such as allowing agents to share locations for a resource penalty (Shi et al. 2019). However this paper still aims for no collisions in the final solution. SIPPS does its best to avoid soft obstacles by looking for a path with the fewest soft obstacle collisions. To do this nodes are expanded by order of a composite priority function that prefers lower soft obstacle collisions, with ties broken by lower f(n), then by higher g(n), and then by most recent node generation. Any soft obstacle collisions that remain in the returned paths are unavoidable in the current state in the search space but are expected to be resolved in future iterations of neighbourhood replanning with a potentially better set of hard obstacles and neighbourhood of agents.

Conflict-Based Search (CBS)

Like in Prioritised Planning, paths of agents outside the neighbourhood comprised hard obstacles however collisions that arose in the conflict tree were also considered hard obstacles. Without the use of soft obstacles, this implementation effectively runs SIPP.

Implementation Details

We implemented LNS2 using Python rather than C++ as the authors did due to our lack of familiarity with the language. We refactored our codebase to implement the various components of the algorithm with optimal time-complexity but may not have optimised everything to the same degree. The authors were not explicit as to what constituted hard obstacles and soft obstacles, but we made an interpretation that fits with their overview description and general process of the algorithm. A few utility functions were adapted from the CMPT417 individual assignment

Experimental Motivation and Setup

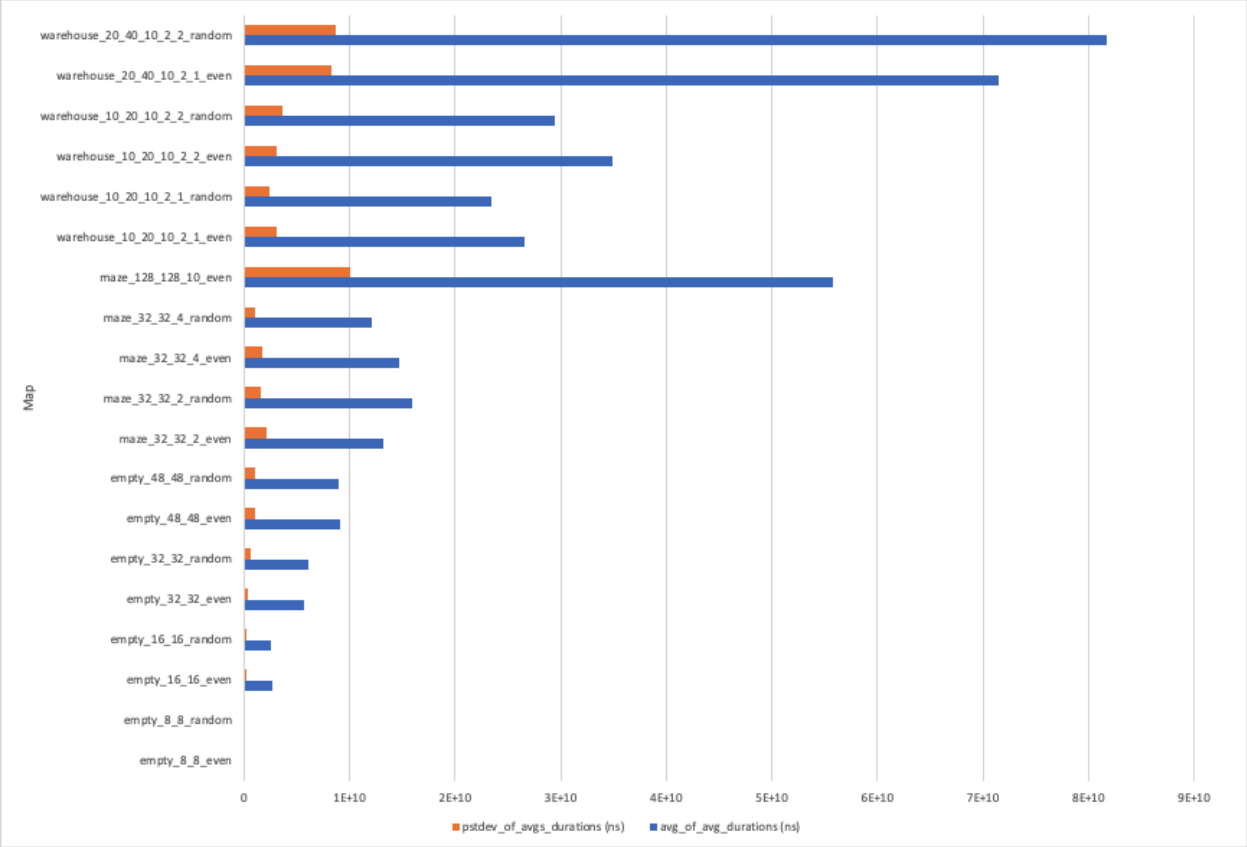
In this project we wanted to achieve two things. First we wanted to challenge ourselves to reproduce the results of a state-of-the-art algorithm. We will look at various measurements of average and standard deviation of execution duration to observe the influence of map features on the performance of the LNS2 algorithm, as well as the statistical distribution of performance of the algorithm on the same scenario due to random factors. While we did not have the resources to perform the scale of experiments shown in the paper, we ran performance tests on a select number of maps from the MAPF benchmark (Stern et al. 2019) to analyse certain map features on speed performance. We conducted these tests on Amazon AWS's EC2 'm4.xlarge' instances with 16GB memory which are the same instances as the authors used.

During the course of studying this algorithm we came up with a second idea. The authors mentioned that in a previous paper the effect of running this algorithm with different neighbourhood sizes produced negligible differences in resulting times, a large neighbourhood resolving more conflicts at once comes trades for it with taking longer to run, and smaller neighbourhoods require more iterations. Motivated by this observation, we wondered if it is feasible to use CBS for replanning on small neighbourhoods. The constant small neighbourhood size avoids the exponential time scaling of bigger neighbourhoods on CBS, and CBS might be able to make better improvements per iteration of replanning. If the time results are not too slow, improved CBS variants might offer a promising opportunity to improve or trade-off the authors LNS2 implementation. The authors also measured the performance of PP and Explicit Estimation CBS with SIPPS (EECBS*) (Li et al. 2021), the previous state-of-the-art suboptimal MAPF solver which vastly outperformed PP. Adapting EECBS* to work as a neighbourhood replanner for ALNS is an immediately interesting prospect. Experiments comparing PP to CBS for neighbourhood replanning in ALNS was done in an i7-8700K 3.7GHz processor with 16GB memory.

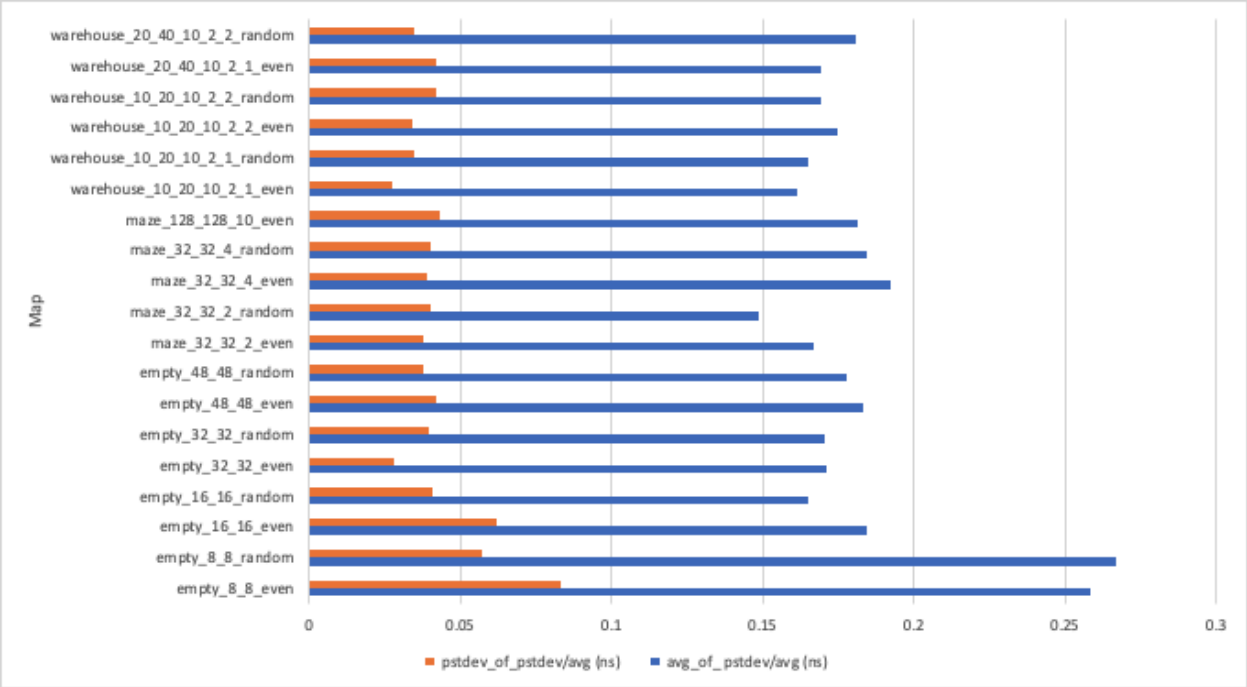
Results

Experiment 1: LNS2 performance (Graph 1, Graph 2)
For each map we ran all 25 of the even scenarios solving for up to 100 agents using a neighbourhood size of 5 and a time limit of 4 minutes. We measured the average and standard deviation of execution times of 15 iterations on the same scenario, and also aggregated the average and standard deviations of those for all scenarios of a map.

Experiment 2: PP vs CBS for neighbourhood replanning (Table 1)
Even-1 scenarios of empty-32-32, maze-32-32–2, room-32-32-4, and den312d maps were tested with 100 agents with a neighbourhood size of 5 and time limit of 2 minutes.Each scenario was tested 10 times and the averages and standard deviations of the path cost, duration, and replanning iterations are reported.



Graph 1. Average and standard deviation of runtime for all even scenarios of each map. Each of 25 even or random scenarios was solved 15 times for a total of 375 iterations per run. Duration is measured in nanoseconds. Corresponding even/random scenarios for a map not shown here did not complete their runs in time before the project deadline.



Graph 2. Average and standard deviation of the variability of LNS2's performance on the same scenario. From the same experimental runs as graph 1. Units are in terms of the relative size of the standard deviation to the average duration of the scenario.

Table 1. Comparative performance of PP/SIPPS vs. CBS/SIPP

Map	Algorithm	Cost avg	Cost stdev	Duration avg	Duration stdev	Replans avg	Replans stdev
room-32-32-4	PP	5552	122	5.62s	1.51s	27	7.8
	CBS	5187	125.1	26.14s	24.82s	40	8.8
maze-32-32-2	PP	7724	482	7.57s	1.21s	15	3.4
	CBS	6812	—-	38.87s	—-	40	—-
empty-32-32	PP	2313	42.44	3.41s	0.73s	20	5.5
	CBS	2247	27.8	4.11s	0.67s	30	7.7
den312d	PP	7257	135.3.	11.72s	0.96s	18	2.5
	CBS	6989	73.7	19.27s	11.27s	29	5.6

Discussion

Map Specific factors

The maps span a range of sizes, which makes direct comparison among them all impossible due to the effect on the runtime of SIPPS, but between maps of the same size we can compare the ability of LNS2 to navigate the collisions induced by the map topology. Within the warehouse maps, wider hallways of size 2 take slightly longer to solve. This may be due to SIPPS needing to generate more nodes for the same linear path adding a consistent time overhead. The standard deviations between width-2 and width-1 warehouse maps are almost exactly the same which supports this idea. The effect is not observed in maze-32-32-4 vs. maze-32-32-2. The major difference is that the topology of hallways in the warehouse map is densely connected in a grid-like fashion, while the topology of a maze is more like a tree graph, whatever increase in time might have been due to SIPPS exploring more areas might be balanced out by other factors related to the tree-graph topology that are not immediately obvious. The empty maps are ordered with almost linear correlation between their times and single dimensional side length. It also appears that even scenarios tend to take longer to solve with the few exceptions of maze-32-32-2 and empty-32-32. There is a motivation to explore and correlate properties of a map's network flow graph to estimate the performance of MAPF algorithms.

Algorithm inherent randomness

There seems to be a consistency in the variability of how long it takes LNS2 to find a solution. All maps except empty-8-8 hover around a standard deviation of 17% of the average duration. The standard deviation of standard deviations also seems to be consistent. Due to the small size of the empty-8-8 map we might suspect that it stands a higher chance at obtaining a close to collision free initial plan than maps of larger sizes. Overall there is nothing unexpected about the randomness of LNS2's performance from a typical Gaussian distribution.

PP vs. CBS

CBS standard deviations for maze-32-32 are unavailable due to it timing out 8 out of 10 iterations. PP manages to generally outperform CBS in speed and replanning iterations, but not in cost. Room-32-32 and maze-32-32 exhibit larger differences in both PP's time and neighbourhood advantage and CBS's cost advantage. PP and CBS have comparable times on empty-32-32 even with CBS's 50% higher replanning count. Den312d exhibits a moderate level of the tradeoff pattern. What's especially noticeable is CBS's large time standard deviations, reaching similar magnitudes as its average in room-32-32 and den312d, and being inferred that for maze-32-32 it may have been even higher. The lowest time for CBS in room-32-32 was 5.9s with 24 replans to 110s with 47 replans. The mismatch between duration and replanning iterations points to the CBS algorithm being responsible for the large spikes in time. This was observed during testing and debugging as well. It is well documented that CBS performs terribly in certain situations. Particularly common is the rectangle symmetry around a cardinal conflict where CBS will explore every possible conflict between two agents crossing paths before it decides to perform a wait action. This and a number of other situations are solved in EECBS and EECBS* (Li et al. 2021). Performing this experiment again with the features used to formulate EECBS* or other optimal CBS variants could be extremely promising given the consistently lower cost of CBS.

Citations

Phillips, M., & Likhachev, M. (2011). SIPP: Safe interval path planning for dynamic environments. *2011 IEEE International Conference on Robotics and Automation*, 5628-5635.

Li, J., Chen, Z., Harabor, D.D., Stuckey, P.J., & Koenig, S. (2022). MAPF-LNS2: Fast Repairing for Multi-Agent Path Finding via Large Neighborhood Search. *AAAI*.

Shi, R., Steenkiste, P., & Veloso, M.M. (2019). SC-M*: A Multi-Agent Path Planning Algorithm with Soft-Collision Constraint on Allocation of Common Resources. *Applied Sciences*.

Stern, R., Sturtevant, N.R., Felner, A., Koenig, S., Ma, H., Walker, T.T., Li, J., Atzmon, D., Cohen, L., Kumar, T.K., Boyarski, E., & Barták, R. (2019). Multi-Agent Pathfinding: Definitions, Variants, and Benchmarks. *ArXiv, abs/1906.08291*.

Li, J., Ruml, W., & Koenig, S. (2021). EECBS: A Bounded-Suboptimal Search for Multi-Agent Path Finding. *AAAI*.