Comparing Multi-Agent Pathfinding Algorithms

James Bie, Winfield Chen, Zhiqi Qiao

Introduction

Multi-agent pathfinding (MAPF) on *k* agents seeks to find a set of *k* non-conflicting paths in a graph from each agent's source vertex to each agent's target vertex. We compare three MAPF algorithms: Increasing Cost Tree Search (ICTS), Enhanced Partial Expansion A* (EPEA*), and Integer Linear Programming (ILP).

Implementation

Integer Linear Programming

We base our ILP implementation off of Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski's "Efficient SAT Approach to Multi-Agent Path Finding under the Sum of Costs Objective," which focuses on a SAT encoding of MAPF. Since SAT and ILP are both NP-Complete problems, they are reducible to each other, and thus we knew it was possible to synthesize the presented SAT encoding with the integer multi-commodity flow formulation discussed in class (MAPF via Reduction slides, "Compact Formulation"), which we will refer to as "the flow formulation" from now on. We did all of the implementations of our flow formulation in Python using the NetworkX graph library, the Pyomo mathematical optimization library, and the GNU Linear Programming Kit (GLPK) solver.

We initially implemented the flow formulation using gadgets to encode vertex and edge collision constraints. However, encoding constraints in this way resulted in a large and unwieldy time-expanded graph (TEG) which required construction each iteration. We decided to make the first improvement to our implementation; encoding constraints the straightforward way as actual ILP constraints in a compact formulation without gadgets.

We encoded edge and vertex collision constraints as additional constraints to our ILP solver instead of as additional vertices and edges as gadgets in the TEG. This reduced the size of the TEG massively, which reduced the number of decision variables optimized by the ILP solver (replacing them with constraints), reducing the amount of time required per solve.

As usual in multi-commodity flow, the decision variables are the amount of flow of each agent's commodity over each edge of the TEG. The objective in our flow formulation is to maximize the incoming amount of each agent's commodity flow at the goal vertex. We note that the usual flow capacity, conservation, sources, and sinks constraints are present as in all multi-commodity flow formulations. In our formulation, total flow capacity of each edge is one, net flow (outgoing flow minus incoming flow) is zero across all vertices for each agent's commodity, the start vertex is the source of one unit of the agent's commodity flow, and the goal vertex is the sink of one unit of the agent's commodity flow. Since the objective and these constraints are common to every multi-commodity flow formulation and are not novel, we only include the code for the vertex and edge collision constraints below, which limit the total outgoing flow at each vertex to be at most one and limit the total flow across pairs of crossing edges (*e_1 = ((i, j, t), (i', j' ,t + 1))* and *e_2 = ((i', j', t), (i, j ,t + 1))* are a pair of crossing edges) to be at most one.

```
model.vertex_collisions = pyo.ConstraintList()
for v in T.nodes:
    if T.succ[v]:
        model.vertex_collisions.add(
            pyo.summation(
                model.x,
                index={(agent, e) for e in T.out_edges(v) for agent in agents}) <= 1)
```

```
model.edge_collisions = pyo.ConstraintList()
for u, v in T.edges:
    if u > v and ((*v[:2], u[2]), (*u[:2], v[2])) in T.edges:
        model.edge_collisions.add(
            pyo.summation(
                model.x,
                index={(agent, x, u[2], y, v[2])
                       for agent, (x, y) in itertools.product(
                        agents,
                        itertools.permutations([u[:2], v[:2]]))}
            ) <= 1
        )
```

This change also meant the TEG construction procedure could be simplified to just the tensor product between the map movement graph (a 2-dimensional 4-connected grid digraph with self-loops and the vertices corresponding to obstacle cells removed) and the path digraph of makespan+1 vertices (*mu*+1 vertices; we refer to the makespan as *mu* in formulae from now on) which represented the directional flow of time.

```
# Create graph
self.map = np.array(map)
self.obstacles = [coords for coords, val in np.ndenumerate(self.map) if val]
self.G_undirected = nx.grid_2d_graph(*self.map.shape)
self.G_undirected.remove_nodes_from(self.obstacles)
self.G = self.G_undirected.to_directed()
self.G.add_edges_from((v, v) for v in self.G.nodes)
T = nx.tensor_product(self.G, nx.path_graph(mu + 1, create_using=nx.DiGraph))
```

This simplified TEG construction was much more elegant (definition of elegance: time expansion as the tensor product of possible moves and the flow of time!) than the

complicated construction procedures required to incorporate gadgets. However, our flow formulation still minimized makespan instead of sum-of-costs. Our second improvement to our implementation is where we combined the theory of the SAT encoding paper with our flow formulation to minimize sum-of-costs instead.

We added the calculation of the individual costs of each agent, which was found using A* search with Manhattan distance heuristic. We then added the calculation of the lower bound on minimum sum-of-costs (*xi_0*) and the lower bound on minimum makespan (*mu_0*).

```
self.individual_costs = [nx.astar_path_length(self.G, start, goal, manhattan)
                         for start, goal in zip(starts, goals)]
self.xi_0 = sum(self.individual_costs)  # Lower bound on minimal sum-of-costs
self.mu_0 = max(self.individual_costs)  # Lower bound on minimal makespan
```

We implemented the iterative search which starts with the sum-of-costs *xi = xi_0* and starts incrementing the value of *xi* being searched for in the solution space. This also increments per iteration the value of *mu* being used to generate the TEG, also known as incrementing the value of *delta*, which is the difference between the searched-for sum-of-costs *xi* that iteration and the lower bound *xi_0*. In each iteration, we generate the TEG and run the GLPK solver on the flows.

```
for xi in itertools.count(self.xi_0):
    print('Searching xi = %d' % xi)
    mu = self.mu(xi)  # Required number of time expansions
    T = self.time_expand(mu)  # Time expanded graph
```

This transition from minimizing makespan to minimizing sum-of-costs was completed with the introduction of an equality constraint on the number of “extra edges,” which are edges in levels deeper in the TEG than what is required by the agent’s individual cost and are not *(goal, t) -> (goal, t + 1)* edges (which represent an agent resting at its goal after having completed its path). The total number of extra edges traversed in the TEG by the agents must equal *delta* in any solution with the searched-for sum-of-costs *xi*.

```
model.extra_edges = pyo.Constraint(expr=pyo.summation(
    model.x,
    index={(agent, u, v)
           for agent, (u, v) in itertools.product(agents, T.edges)
           if u[2] in range(self.individual_costs[agent], mu)
           and not u[:2] == v[:2] == goals[agent]}
) == self.delta(xi))
```

It is here that we actually found an issue with the SAT encoding paper. In the instance *test_32.txt* (one of the instances provided in the individual project), GLPK returned a solution of sum-of-costs 31 while there exists a solution with sum-of-costs 30. The issue is found in the paper’s definition of extra edge: “edges in TEGs corresponding to wait actions at the goal are not marked as extra.” (Section 3.3 Page 3) However, in the solution found for *test_32.txt* by GLPK, an agent arrives at its goal, remains there for some time, and then moves aside to allow another agent to pass before returning to its goal. Here we see the flaw of defining each *(goal, t) ->*

(goal, t + 1) edge as non-extra: an agent may not have completed its path once it traverses such an edge; it may move off its goal at some point afterwards, and thus this non-extra edge may become an extra edge in that case only. But we don't know if this case applies until after a solution to the ILP is found!

We note that this is not an incredibly significant issue theoretically because the sum-of-costs 30 (true optimal) solution is still a solution to the ILP (and the SAT encoding). However, it is a big issue for any practical implementation based on the SAT encoding paper. GLPK's ILP branch-and-bound solver branches upon variables in a certain order as determined by the Tomlin-Driebeek heuristic (the GLPK LP relaxation simplex solver also has a certain order in which it pivots), and this order determines which solution is returned by GLPK. A branch-and-bound SAT solver on the SAT instance would also have this property.

In this scenario, there is more than one optimal solution to the ILP satisfying the *delta* extra edges constraint, yet not all of them are true minimum sum-of-costs solutions and GLPK happens to return one of the undesired solutions. We note that if this were an LP, we could access other optimal solutions from the simplex algorithm by pivoting variable columns in the Tucker tableau with a residual cost of zero, however, this cannot be done in an ILP (we would need to repeat the NP-hard branch-and-bound step which is slow since at worst we would need to enumerate all optimal solutions, i.e. this is a problem in #SAT complexity class, and it is already a proven result that knowing one SAT solution does not assist in enumerating all SAT solutions [recall ILP and SAT are really the same problem in complexity theory]). Thus we were unable to get the true minimum sum-of-costs solution from GLPK into Pyomo for *test_32.txt*. We did not find any commentary in the SAT encoding paper which discussed this issue.

Finally, our last improvement to the flow formulation was pruning the TEG by removing vertices that were not helpful to the agents objectives of going from their starts to their goals. The TEG is a directed acyclic graph (DAG) so finding ancestors and descendants is easy. We removed vertices in the TEG which were not descendants of *(start, 0)* (we keep *start* itself) and ancestors of *(goal, mu)* (we keep *goal* itself) for at least one agent. This is a similar simplification to converting a TEG for an agent into a multi-value decision diagram (MDD) for that agent (the agent's MDD is a subgraph of the agent's TEG).

```
keep = set(itertools.chain.from_iterable(
    (nx.descendants(T, start) | {start}) & (nx.ancestors(T, goal) | {goal})
    for start, goal in zip(zip(self.starts, itertools.repeat(0, len(self.starts))),
                           zip(self.goals, itertools.repeat(mu, len(self.goals))))))
T.remove_nodes_from(T.nodes - keep)
```

In fact, the resulting pruned TEG is the union of all agents' MDDs (the MDD for an agent is precisely the subgraph of the TEG for the agent induced by the descendant-of-start-and-ancestor-of-goal procedure). By pruning unhelpful nodes (nodes which could not possibly lie on a start-to-goal path for any agent) from the TEG, we further

reduced the solve time of the flow formulation by eliminating even more decision variables and constraints. This final improvement completes our flow formulation.

Enhanced Partial Expansion A*

When solving instances of problem domains that feature a large branching factor, A* may generate a large number of nodes whose cost is greater than the cost of optimal solutions, those nodes are called surplus nodes. Generate surplus nodes and adding them into the open list will increase the time and memory usage of the search. To save memory, a variety of A* is introduced called Partial Expansion A* (PEA*). When expanding a node n, PEA* generates all of its children and puts into OPEN only the children with f = f(n), the rest of children will be discarded. n is reinserted in the OPEN list with the f-cost of the best-discarded child. This guarantees that surplus nodes are not inserted into OPEN.

EPEA* is an enhanced version of PEA*. EPEA* generates only those children nc with f(nc) = f(n). In contrast to PEA*, the other children are not even generated. This is enabled by using a-priori domain- and heuristic-specific knowledge to compute the list of operators that lead to the children with the needed f-cost without actually generating any children. This knowledge and the algorithm for using it form an Operator Selection Function (OSF). For a given node expansion, an OSF returns the set of children with f(nc) = f(n) and the smallest f-value among the children with f(nc) > f(n) as $F_{next}(n)$. If there's no other child with f(nc) > f(n), then set $F_{next}(n)$ equal to infinite. Then reinsert n in the OPEN list with the $F_{next}(n)$ value if it is not infinite. If $F_{next}(n)$ is equal to infinite, that means there are no discarded children so this node can be closed and inserted into the CLOSE list.

Pseudocode for EPEA*:

```
Generate the start node n_s
Compute h(n_s) and set F(n_s)←f(n_s)←h(n_s)
Put ns into OPEN list
While OPEN is not empty:
  get n with the lowest F(n) from OPEN
  If n is the goal then exit
  get set N and F_next(n) from OFS(n)
  If N is empty:
          set F(n)=F_next(n) and reinser n into OPEN
      continue
  For all child in N:
      set g(child)=g(n)+cost(n, child)
      set f(child)=h(child)+g(child)
      If child not in OPEN and CLOSE:
          set F(child)=f(child)
```

Insert child into OPEN list
If Fnext(n)=infinite:
Put n into CLOSE list
Else:
set F(n)=F_{next}(n) and reinser n into OPEN
EXIT

Increasing Cost Tree Search

Our implementation of the increasing cost tree search was based on the paper written by Guni Sharona, Roni Sterna, Meir Goldenberg and Ariel Felnera, “The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding,” as well as the notes discussed in class. Our ICTS algorithm is split into a high-level and a low-level. The high level is a tree whose root node starts as a combination of the minimum path lengths needed for each agent to reach their goal node in the absence of other agents. These minimum path lengths were found using A* search with a heuristic that was generated using Dijkstra’s algorithm. Each child of a node in the high-level tree will increment one of the agent’s minimum path by 1 and the resulting child will be stored in a hash table. This hash table will be checked every before generating any children to prevent duplicate nodes from being created. Finally, the tree will be explored with breadth first search until a solution is found.

Once the high-level tree node is selected our algorithm moves to the low-level algorithm that determines if a solution can be reached. Our algorithm begins by generating an MDD graph by breath-first searching through the map for as many steps as the cost passed down through the high-level node. Each section on the map can have 5 children (moving up, down, left, right and weighting). Each node in the MDD graph will keep track of its parent, children, cost and location.

Resulting MDD graphs will be combined by traversing through both MDD graphs level by level. All nodes on the same level were cross-product and added as a child of the previous level based on the parent information stored in each node. If a node had two agents occupying the same location it was pruned as that would be a conflicting state. Furthermore, if the location of the (parent, child) of one agent matched (child, parent) of another it would indicate an edge collision and was also pruned. Nodes were once again added to a hash table to prevent duplication. In our algorithm MDD graphs were constructed from scratch every time. If more time was permitted, then recycling old MDD graphs can save resources for future MDD graphs. Once all the MDD graphs have been combined into one final MDD graph (with conflicts pruned) we found our solution by depth first searching the graph until a goal node (for all agents) was found. Depth first search was chosen as the goal node should lie in the leaves of the MDD graph if it exists as the MDD graph expands in a stepwise fashion. This means that if a solution exists depth-first searching may find it

more quickly. Once a goal node was found, it was a matter of tracing a route back to the root node by following a series of parent nodes and returning that path.

Pseudocode

```
For each agent:
        Paths = Find shortest path from start to goal for each agent ( a* search)

#High-level algorithm

Searchlist = [paths]
duplicateList = {}
While solution not found:
        Search = searchList.pop[0] #search first node in the search list

#low-level algorithm
        For each agent:
            ListOfMDD = GenerateMDD(startnode, cost based off of path)
        mainMDD = listofMDD[0]
       For I in   range(agent-2):
            mainMDD = FuseMDD(mainMDD, listOfMDD[i+1)
    Path = depthfirstsearch(mainMDD, target = goals)
    If pathExists:
        Return path
#High-level algorithm to add new children to search for that increment different agent path lengths
    For agent in (number_of_agents):
        Child = copy. Search
        Child[agent] += 1
        If child is not in duplicatelist:
            searchList.append(child)
            duplicatelist.add(child)
```

Methodology

Integer Linear Programming

We compared the total GLPK solve time of the flow formulation to the Conflict-Based Search (CBS) implementation from the individual project. We run this comparison over all provided *test_*.txt* instances of the individual project and run a focus trial on the most difficult instance, *test_47.txt*, which has 7 agents on an 8x8 grid with a minimum sum-of-costs value of 65. We compare this focus trial with *test_4.txt*, which has only 5 agents also on an 8x8 grid and with a minimum sum-of-costs value of 32.

We intend to use this comparison to find the effect increasing the number of agents has on the runtime of ILP and CBS, and whether ILP is faster than CBS as the number of agents increases, which we suspect to be the case.

Enhanced Partial Expansion A*

We want to investigate how much memory usage EPEA* saved compared to A*. To find the answer, we did a benchmark on different instances using CBS with A* and CBS with EPEA*. We did a benchmark on test_1, test_41, maze-32-32-2.map from https://movingai.com/benchmarks/mapf.html and test_47. We did benchmark on different kinds of map with different number of agents to see the influence of the algorithm. We track the CPU running time, expanded nodes number and generated nodes number.

Increasing Cost Tree Search

We will be investigating the resource usage, in terms of time and nodes expanded, of Increasing Cost Tree Search compared to Conflict Based Search, EPEA* and Integer Linear Programming. We want to see Increasing Cost Tree Search's performance on larger maps as well as with more agents. This may end up being important as Increasing Cost Tree Search has to generate the MDD for each agent and cross-product each level node. This is expensive and could offset any savings that the algorithm provides as well as taking up a lot of space to store the value.

Environment

Integer Linear Programming

We run our ILP-CBS comparisons on *asb9838-1-a01.csil.sfu.ca*, a machine in the Computing Science Instructional Laboratory (CSIL) with an 8-core Intel Core i7-6700 (3.40 GHz) and 32 GB of memory running Ubuntu 18.04.

Enhanced Partial Expansion A*

We run our EPEA* in a Windows 10 environment, with two 20-core Intel Xeon CPU E5-2680 v2 (3.20GHz) and 64 GB of memory.

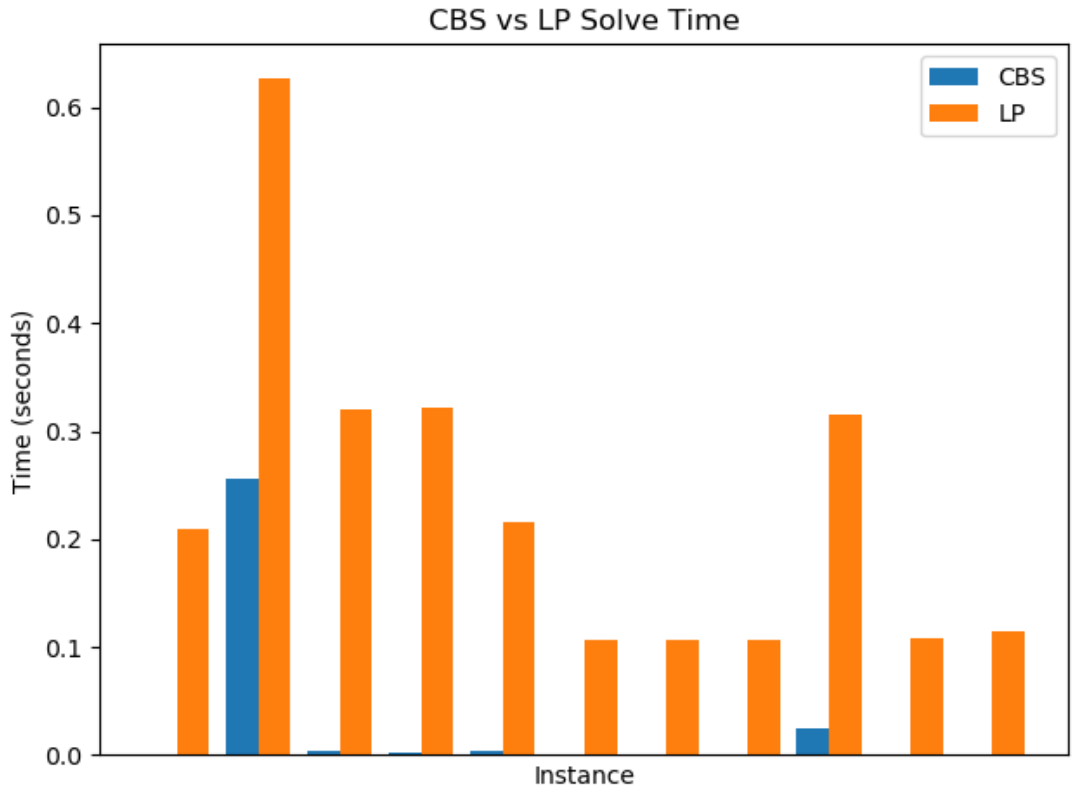
Increasing Cost Tree Search

We run ICTS on MACOS Catalina Version 10.15.3 with 16 GB of memory and an intel HD Graphics 630 and a 2.9 GHz Quad-Core Intel Core i7. The algorithms were run on Python 3.

Results

Integer Linear Programming

The total solves time of all *test_*.txt* instances for the ILP flow formulation is 27.62 seconds, whereas for the CBS implementation it is 212.19 seconds. We note that the speed advantage of ILP compared to CBS becomes especially pronounced as the number of agents grows, as we can see with our focus trial on *test_47.txt*, which we recall has 7 agents. In this case, the flow formulation takes 6.18 seconds and CBS takes 210.95 seconds. However, ILP loses its speed advantage in problems with a small number of agents. In our comparison focus trial on *test4.txt*, which we recall includes only 5 agents, ILP takes 0.32 seconds of solve time whereas CBS takes 0.01 seconds. This makes sense; the multi-commodity flow formulation gains a commodity for each additional agent (adding more flow variables and constraints) which scales better than the additional work done by CBS on the introduction of an additional agent, however the size of a TEG may remain large even when the number of agents is small (whereas CBS does not require a TEG). To illustrate this point, we attach a plot of performance across the *test_3*.txt* instances, which all have 5 agents and are therefore advantageous for CBS:



Enhanced Partial Expansion A*

We did a benchmark on test_1, which has a sum of cost value of 41. It took a CPU time of 0.11s, expanded 145 nodes, and generated 280 nodes using A* with CBS. When using CBS with EPEA*, it took a CPU time of 0.15s, expanded 42 nodes, and generated 83 nodes.

We also did a benchmark on test_41, which contains 6 agents, an 8x8 grid, and the sum of costs is 45. When using A* In CBS search, it took 0.50s CPU time, expanded 407 nodes, and generated 813 nodes. When using EPEA* In CBS search, it took 2.04s CPU time, expanded 407 nodes, and generated 813 nodes.

EPEA* did save some memory on expanded and generated nodes. However, it took more CPU time to run. For instances with fewer agents but a complicated map, for example, maze-32-32-2.map in the provided benchmarks, A* and EPEA* both perform very well. However, in an instance with many agents such as test_47, EPEA* does not perform very well on running time. Although it saved some memory on expanded and generated nodes, it wasted too much time on generating OFS and checked duplicates.

Increasing Cost Tree Search

Unfortunately we were not able to create an ICTS that was able to work with more than 2 agents. However, based on literature ICTS will be superior to A* when there is not a big difference between the distance travelled when an agent is alone and the distance travelled with other agents. This makes sense as the high-level tree of ICTS would have more iterations that it has to get through as it travels through with breadth-first. One pathological case from this knowledge is if two agents need to swap places but cannot because of an edge conflict. If the only place they can rearrange their order is far away then ICTS will be inefficient. This means if a map is crowded (high agents and small map size) with a lot of conflicts or the pathological case then ICTS will perform poorly compared to A* but it performs better otherwise. Our ICTS did not employ many of the ICT pruning techniques and ways to optimize the MDD creation. Therefore, I also expect our results to be worse compared to A* due to the expensive nature of creating MDDs.

# of agents	Obstacles	A*	EPEA*	ILP
3	No	0.02	0.03	0.229630
3	Yes	0.02	0.00	0.103077
5	No	0.02	0.04	0.756837
5	Yes	0.01	0.00	0.117096
8	No	132.40	204.43	21.199285
8	Yes	6.69	12.12	13.565662

Time in Seconds for the three algorithms to find an optimal path in a 6x6 grid. Obstacle is a wall that travels down the (x,4) column with the exception of a gap at (1,4).

Conclusions

We conclude that ILP has better time performance than CBS on MAPF instances with large numbers of agents.

EPEA* will significantly save memory on expanded nodes and generated nodes on most of the cases, but it will take more running time than A*.

Bibliography

Pavel Surynek, Ariel Felner, Roni Stern, and Eli Boyarski, “Efficient SAT Approach to Multi-Agent Path Finding under the Sum of Costs Objective,” https://docs.wixstatic.com/ugd/749b4b_c7d1fe5a70a84dbe8f52a6695e45457b.pdf

M. Goldenberg, A. Felner, R. Stern, G. Sharon, N. Sturtevant, R. C. Holte, J. Schaeffer, “Enhanced Partial Expansion A*. ” https://www.jair.org/index.php/jair/article/view/10882/25958

Guni Sharona, Roni Sterna, Meir Goldenberg, Ariel Felnera, “The Increasing Cost Tree Search for Optimal Multi-agent Pathfinding” https://www.ijcai.org/Proceedings/11/Papers/117.pdf