Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)

Lazy CBS: Implicit Conflict-Based Search Using Lazy Clause Generation

Graeme Gange, Daniel Harabor, Peter J. Stuckey
Faculty of Information Technology,
Monash University, Melbourne, Australia
{graeme.gange,daniel.harabor,peter.stuckey}@monash.edu

Abstract

Conflict-based Search (CBS) is a effective approach to optimal multi-agent path finding. However, performance of CBS approaches degrade rapidly in highly-contended graphs with many agents. One of the reasons this occurs is that CBS does not detect independent subproblems; i.e. it can re-solve the same conflicts between the same pairs of agents up to exponentially many times, each time along a different branch. Constraint programming approaches with nogood learning avoid this kind of duplication of effort by storing nogoods that record the reasons for conflicts. This can exponentially reduce search in constraint programming. In this work, we present Lazy CBS, a new approach to multi-agent pathfinding which replaces the high-level solver of CBS with a lazily constructed constraint programming model with nogoods. We use core-guided depth-first search to explore the space of conflicts and we detect along each branch reusable nogoods which help to quickly identify feasible solutions. Our experiments show that Lazy CBS can significantly improve on the state-of-the-art for optimal MAPF problems under the sum-of-costs metric, especially in cases where there exists significant contention.

Introduction

Multi-agent Pathfinding (MAPF) is a planning problem which asks us to coordinate the movements of a team of agents: from a set of unique starting locations to a set of unique target positions all while avoiding collisions. This problem appears in a variety of different application areas including warehouse logistics (Wurman, D'Andrea, and Mountz 2008), office robots (Veloso et al. 2015), aircraft-towing vehicles (Morris et al. 2016) and computer games (Silver 2005). Often the the environment in which agents operate is given as an undirected graph, such a grid. Common objectives then include minimising the total arrival time of all agents (aka. sum-of-costs) and minimising the arrival of the last agent at its target location (aka. makespan). In each of these common settings MAPF is known to be NP-hard (Yu and LaValle 2013) to solve optimally. Interest in the problem has nevertheless generated a wide variety of methods including optimal, suboptimal and and bounded suboptimal techniques. A recent survey of the area is given in (Felner et al. 2017).

In this work we are interested in computing optimal plans for MAPF problems on undirected graphs and under sum-of-costs. A dominant family of algorithms appearing in this setting is Conflict-based Search (Sharon et al. 2015). CBS-like algorithms divide the MAPF problem into two parts: a *low-level* search – which finds optimal paths for individual agents – and *high-level* search, which both tracks and resolves conflicts between pairs of agents. Both the high- and low-level solvers of CBS are typically implemented as A^* or else other similar best-first techniques.

The efficiency of the CBS framework is highly dependent on having accurate heuristics: unfortunately, while this is (initially) true for the low-level solver, the high-level solver often discovers objective changes quite late in the search process. So while CBS-based approaches often perform quite well on sparse problems with low contention, success rates can drop dramatically as the number of agents and rate of contention increase. A key limitation of the CBS approach is the inability to learn information across nodes.

Example 1 *Consider the four-agent pathfinding problem illustrated in Figure 1(a). This consists of two independent subproblems, each requiring one of the two agents to wait before crossing. CBS chooses one of the two conflicts, e.g. between a_1 and a_2, and the search starts expanding children, eventually discovering that every solution to this conflict causes the objective to increase. Figure 1(b) shows the tree generated by CBS in this case. The problem is that the conflict between a_3 and a_4 exists in every node generated thus far and under every child CBS will generate the tree shown in Figure 1(b). Overall CBS will generate 110 high-level nodes before it can find an optimal solution, with each generated node resulting in a call to the low-level pathfinder.*

For an alternative perspective consider a hypothetical constraint programming model for MAPF as follows: there exist variables $\mathtt{p}[l,t]$ where $\mathtt{p}[l,t] = i$ indicates that agent a_i *is permitted to* occupy location l at time t. There also exist variables c_i which represent the cost of the path for agent i. The objective in this model is to minimize the sum of all c_i, subject to constraints:

- for each agent, there exists a path with cost at most c_i and,
- at most one agent is at a given location at any time and,
- at most one agent can transition between any pair of adjacent cells at any time.

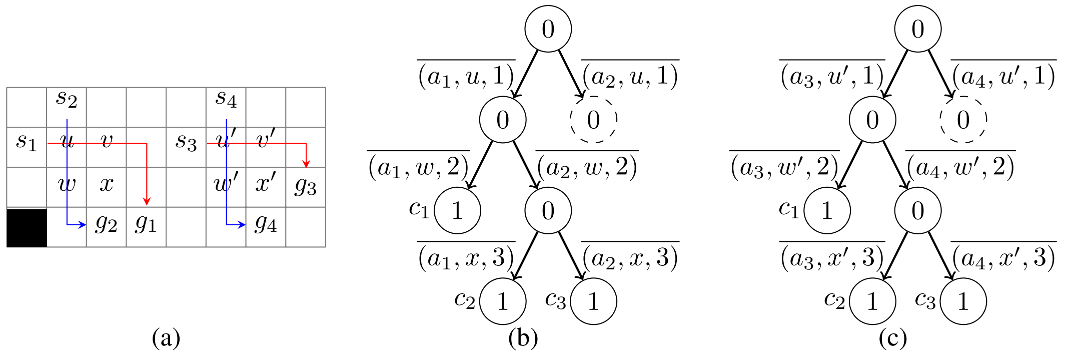


Figure 1: (a) MAPF problem containing independent conflicts, (b) partial search tree for (a_1, a_2), and (c) partial subtree for (a_3, a_4) conflicts. In CBS, the tree shown in (c) will be fully expanded below each leaf in (b).

Indeed, (Surynek 2012) describes a SAT-based makespan-optimal MAPF solver using a very similar encoding. The pure SAT-based model performs very well on small graphs with very high occupancy, but has trouble with larger graphs and other objectives (i.e. sum-of-costs).

Rather than add all constraints and all variables to the initial model however, we propose to use this formulation *lazily*. Like CBS, our approach distinguishes between high-level and low-level search and like CBS our low-level solver plans paths for individual agents using time-expanded A*. Unlike CBS our high-level search does not branch on detecting a conflict between a pair of agents. Instead, we add the violated constraint to our partial constraint model and we rely on *core guided search* (Andres et al. 2012) to find a feasible plan with minimum-cost.

Where high-level CBS explores the space of all pairwise conflicts using a best-first strategy, our core-guided search uses a depth-first traversal. Where high-level CBS keeps a set of explicit constraints applicable only for the current node, our core-guided search constructs a database of nogoods: learned constraints that explain where the search previously failed and under what conditions. Similar in spirit to the way a CLOSED list helps A* to avoid duplicated effort, a database of nogoods helps our core-guided search to avoid re-solving subproblems previously seen.

Example 2 *Consider the example shown in Figure 1(a). Core-guided search will lazily add variables to represent which agent is permitted to use locations u, v, x and (possibly) w as it attempts to resolve the conflict between agents a_1 and a_2. Once it determines that it is not possible for both agents to take their shortest path the solver learns the nogood $\langle c_1 \geq 6\rangle \vee \langle c_2 \geq 5\rangle$ and a subsequent solution where e.g. agent a_1 waits for a_2. Next, the solver will explore the conflict for agents a_3 and a_4, eventually learning the nogood $\langle c_1 \geq 5\rangle \vee \langle c_2 \geq 5\rangle$ and the subsequent solution where e.g. a_3 waits for a_4. Critically, by the nature of nogood learning, even if the search interleaves conflict resolutions between the two agent pairs, since the conflicts are independent, the same learning will result.*

Preliminaries

Problem: We are given as input an undirected graph $G = (V, E)$ which represents an operating environment where V is a set of locations (equiv. nodes) and E is a set of edges that serve to transition from one location to another. We assume there exists a function $\textsc{adjacent}(G, l)$ which returns the nodes adjacent to node l.

There exists in the environment a team of agents $\mathcal{A}$ such that each $a_i \in \mathcal{A}$ has a unique start location $s_i \in \mathcal{S}$ and a unique target location $t_i \in \mathcal{T}$ with $\mathcal{S}, \mathcal{T} \subseteq V$. Time is discretised into *steps* of unit duration and at each step agents can move from one vertex to the next at a cost of 1. Agents can also *wait* at their current location at a cost of 1 and they can wait at their target location at a cost of 0 but only if they do not move again after arriving.

While moving agents are subject to a set of constraints. The first constraint is that no more than one agent can occupy a given vertex at any one time. We model this as $\overline{(a_i, l, t)}$ which states that agent a_i is forbidden from using location l at timestep t. In addition, no more than one agent can transition between the same pair of vertices at the same time. We model this as $\overline{(a_i, l_i, l_j, t)}$ which says that agent a_i cannot transition from location l_i to location l_j at timestep t.

A feasible solution to the problem is a set of paths that transitions each agent from its start location to its target position while respecting all constraints. An *optimal* solution is one that minimises the total arrival time of all agents. This objective is also known as *sum-of-costs*.

Constraint Programming: Constraint Programming (CP) is a generic approach to solve constraint satisfaction problems. Let $l..u$ define the *range* of integers $\{d \mid d \in \mathbb{Z}, l \leq d, d \leq u\}$. Given an integer variable x we can write *atomic constraints* that constrain x of the form $\langle x \leq d\rangle$, $\langle x \geq d\rangle$, $\langle x = d\rangle$, and $\langle x \neq d\rangle, d \in \mathbb{Z}$. An atomic constraint holds if the value assigned to x satisfies the condition. For our purposes a constraint satisfaction problem (CSP) $P = (\mathcal{V}, \mathcal{D}, C)$ is defined by a finite set of (integer) variables $\mathcal{V}$, an initial *domain* $\mathcal{D}$ which is a set of atomic constraints over variables $\mathcal{V}$, and a set of constraints C. We can equivalently view $\mathcal{D}$ as a mapping from $v \in \mathcal{V}$ to the set of values it is permitted to take. A solution to a CSP is a mapping from each $v \in \mathcal{V}$ to some value d that satisfies all atomic constraints in $\mathcal{D}$, such that each constraint $c \in C$ is satisfied.

Example 3 *When we construct the model for the problem in Figure 1, the only variables present are the path cost variable $c_1, \ldots, c_n$, with initial domain $D_0 = \{c_i \mapsto [0, M] | i \in 1 \ldots 4\}$ for some sufficiently large upper bound M on individual path lengths. D_0 can be equivalently viewed as the conjunction of atomic constraints $D_0 = \bigwedge_{i=1}^{4} \langle c_i \geq 0 \rangle \wedge \langle c_i \leq M \rangle$.* $\square$

CP Execution: Constraints in CP solvers are implemented by propagators that reason by examining the current domain and inferring new information that must hold in that domain. Let $AC(P)$ be the set of atomic constraints for all $v \in \mathcal{V}$. A *propagator* for constraint c is a function from sets of domains to sets of atomic constraints. $p_c : \mathcal{P}(AC(P)) \to \mathcal{P}(AC(P))$. It returns a set of atomic constraints $p_c(\mathcal{D})$ such that $c \wedge \mathcal{D} \to a, a \in p_c(\mathcal{D})$. Constraint programming (CP) solvers repeatedly apply a set of propagators to an initial domain $\mathcal{D}_{init}$ adding any new atomic constraints inferred to the domain until no further progress can be made. If the propagator infers a contradictory set of atomic constraints then the CSP has no solution. If all variables can only take a single value given the current domain $\mathcal{D}$, the problem has been solved (the unique values define a solution). Otherwise the solver *branches*, splitting the current CSP $(\mathcal{V}, \mathcal{D}, C)$ into two subproblems using an atomic constraint a where $D \not\to a$, $D \not\to \neg a$ to obtain two subproblems $(\mathcal{V}, \mathcal{D} \cup \{a\}, C)$ and $(\mathcal{V}, \mathcal{D} \cup \{\neg a\}, C)$ which it explores recursively (in a depth-first manner).

Example 4 *Clearly, the initial domains in Example 3 could be tightened: no agent can achieve a shortest path of length* 0*. After setting up the problem, the propagators run to find the shortest paths for each agent, tightening the domain to $\{c_1 \mapsto [5, M]\} \cup \{c_i \mapsto [4, M] \mid i \in 2 \ldots 4\}$, or equivalently $\langle c_1 \geq 5 \rangle \wedge \bigwedge_{i=2}^{4} \langle c_i \geq 4 \rangle \wedge \bigwedge_{i=1}^{4} \langle c_i \leq M \rangle$.* $\square$

Lazy Clause Generation: Lazy clause generation (LCG) solvers (Ohrimenko, Stuckey, and Codish 2009) augment constraint programming with *conflict-directed clause learning*, which allows the solver to avoid repeatedly exploring infeasible subproblems. In order to do so propagators must be augmented to record the reasoning they perform by providing *explanations*: whenever a propagator for constraint c infers a new atomic constraint inference a under current domain D, i.e. $a \in p_c(D)$, it must also return an explanation $E \to a$, where E is a conjunction of atomic constraints such that $D \to E$, and $c \wedge E \to a$. For more details about lazy clause generation solvers see (Ohrimenko, Stuckey, and Codish 2009).

Core-guided optimization: To solve optimization problems, constraint programming solvers typically rely on branch-and-bound, finding a solution then adding a constraint to restrict search to look for better solutions. Unfortunately, as the constraints communicate only through domains, CP solvers often perform poorly when proving optimality over additive objectives: we can often achieve the lower bound of any one objective component (by sacrificing the others), and we cannot update the objective bound until almost all the component bounds change. Core-guided approaches to optimization can overcome this weakness.

Essentially, core-guided approaches begin by fixing all objective components to their minimum values, and search for a solution to the resulting problem. If this succeeds, we have clearly found the optimum. If the solver concludes that no such solution exists, it returns a *core*: a (not necessarily minimal, but hopefully small) subset of objective components which cannot *collectively* take their minimum values. The solver then relaxes the objective value – but only with respect to variables which appear in the core – and re-solves, repeating this process until a solution is found.

Different core-guided optimization methods differ mostly in how the interaction between successive cores and the objective is handled. The two most fruitful approaches are implicit hitting-set approaches (such as MaxHS (Davies and Bacchus 2011; 2013)), and OLL (Andres et al. 2012).

We shall assume a constraint programming solve function $\text{SOLVE}(C, \mathcal{D}_{init}, A)$ which solves a constraint problem $C \wedge \mathcal{D}_{init} \wedge A$ with constraints C, initial domain $\mathcal{D}_{init}$ and assumptions (atomic constraints) A, either returning an empty set meaning that $C \wedge \mathcal{D}_{init} \wedge A$ is satisfiable, or a subset $S \subseteq A$ such that $C \wedge \mathcal{D}_{init} \to \neg \forall_{c \in S} c$, i.e. its not possible for all assumptions in S to simultaneously hold. Extending nogood learning solvers to support this assumption interface is straightforward (Eén and Sörensen 2003).

Implicit CBS with lazy clause generation

In this section, we present an alternate high-level solver for conflict-based search. Rather than exploring an explicit search tree using A*, we instead maintain an *implicit* representation of the set of feasible conflict resolutions.

Figure 2 gives an overview of the algorithm. We repeatedly call the solver to obtain a solution to the constraints observed so far, with additional *assumptions* that each cost variable takes its minimum value. For each agent a_i, the solver contains a *propagator* which updates its path cost c_i with respect to the constraints enforced. If the solver detects unsatisfiability, it also returns a *core*: a subset of the given assumptions that are collectively infeasible. We then use the core to update the lower bound on the objective. If it finds satisfiability, we check the *alleged* solution for conflicts: if none are found, we have found the optimal plan (with respect to sum-of-costs). If a conflict is found, we introduce a fresh decision variable to resolve the conflict into the solver, and re-solve.

Constraints and constraint solving

Consider what happens when the solver finds a candidate plan where agents a_1 and a_2 both occupy location l at time t. In classical CBS variants, the high-level solver generates two successors: one with constraint $\overline{(a_1, l, t)}$, and the other with constraint $\overline{(a_2, l, t)}$.

```
1: function LAZY-CBS($m$,$[a_1, \dots, a_n]$)
2:     $s \leftarrow$ INITIALIZE-SOLVER($m, [a_1, \dots, a_n]$)
3:     $P \leftarrow$ CURRENT-PLAN($s$)
4:     $L \leftarrow \{c_i i \mapsto |P[i]| \mid i \in 1 \dots n\}$
5:     $lb \leftarrow \sum_{i=1}^{n} L[i]$
6:     $assumps \leftarrow \bigwedge_i \langle c_i \leq L[c_i] \rangle$
7:     while HAS-CONFLICT($P$) do
8:         $conflict \leftarrow$ GET-CONFLICT($P$)
9:         $s \leftarrow$ PROCESS-CONFLICT($s$, $conflict$)
10:        while SOLVE($s$, $assumps$) $\neq \emptyset$ do
11:            $core \leftarrow$ SOLVE($s$, $assumps$)
12:            $s, lb, L \leftarrow$ PROCESS-CORE($s$, $core$, $lb$, $L$)
13:            $assumps \leftarrow \bigwedge_{c_i \mapsto l_i \in L} \langle c_i \leq l_i \rangle$
14:        $P \leftarrow$ CURRENT-PLAN($s$)
15:    return $lb, P$
```

Figure 2: High-level overview of the Lazy-CBS algorithm.

In Lazy-CBS, we instead rely on the solver to manage our exploration of the search space. We introduce a variable $\mathtt{p}[l, t]$ which indicates *which agent* is permitted to use location l at time t. When $\mathtt{p}[l, t]$ is set to some value *other* than i, an obstacle is added to the local map for agent a_i, and the corresponding pathfinder is woken to (possibly) update the lower bound of c_i. Now the solver, in order to find a solution, will have to set this variable to a value, which will permit at most one agent to use location l at time t. Note that $\mathtt{p}[l, t] \neq a_i$ is equivalent to $\overline{(a_i, l, t)}$.

The solver, SOLVE, we invoke is simply a lazy clause generation solver, with the assumptions A and constraints $\mathtt{path}(a_i, c_i, \mathtt{p})$ for each agent a_i. This constraint finds a shortest path for agent a_i from its start to its goal, assuming the constraints described by the $\mathtt{p}$ variables, and updates the lower bound on the c_i variable that records the length of the a_i's path. The propagator is incremental in the sense that it stores a current incumbent path, and if the $\mathtt{p}$ variables still allow this path it does nothing, otherwise it re-plans in the same manner as CBS (Sharon et al. 2015). During the solver's search there is no interaction between the paths of different agents.

The solver can terminate in two ways. If it fails, then it will generate a unsatisfiable core $core$, being a subset of the assumptions, this is a standard feature of the way solvers with assumptions work (Eén and Sörensen 2003). It will then add variables and constraints using PROCESS-CORE and define a new set of assumptions, following an OLL-based approach to core-guided search (Andres et al. 2012). We then re-invoke the solver.

If the solver succeeds then we check whether the paths held by the propagators (CURRENT-PLAN) are mutually consistent (HAS-CONFLICT). If the plan is free of conflicts, it is necessarily an optimal plan. If there is a conflict we choose one (GET-CONFLICT) and introduce a new $\mathtt{p}$ variable to resolve it (PROCESS-CONFLICT). The search is restarted in order to find a solution where the new variable is given a value. Note that this redoes lots of computation, and we could improve our method by being able to add restart the solver directly from the previous solution, this has not yet been implemented. The recomputation is ameliorated since all nogoods discovered in earlier searches are stored we will not redo any failed computation, and using solution-based phase saving (Demirovíc, Chu, and Stuckey 2018) we will direct search to be close to the previous solution, avoiding unnecessary changes to the incumbent paths held in the `path` propagators.

Let us illustrate the operation of Lazy CBS, using our example from Figure 1.

Example 5 *Initially, we start with a solver containing only the four pathfinding propagators* $\mathtt{path}(a_i, c_i, p), i \in 1..4$ *and four cost variable* $c_1, \dots, c_4$*. There are initially no* $\mathtt{p}$ *variables. Running each pathfinder independently, we obtain the initial (infeasible) plan shown in Figure 1(a). We add assumptions* $assumps = \{\langle c_1 \leq 5 \rangle, \langle c_2 \leq 4 \rangle, \langle c_3 \leq 4 \rangle, \langle c_4 \leq 4 \rangle\}$*. We check the plan for conflicts, and find the first conflict to be at time* 1 *at* u*.*

To resolve the conflict at $(u, 1)$*, we introduce a fresh variable* $\mathtt{p}[u, 1]$*, and inform the propagators for* a_1 *and* a_2 *of the new potential obstacle. We then ask the solver for a new candidate solution.* $\mathtt{p}[u, 1]$ *is unfixed, so the solver chooses a value; let us assume* 1*. As* $\mathtt{p}[u, 1] \neq 2$*, the obstacle* $(u, 1)$ *is added to the map for agent* a_2*, which is then re-planned. This yields a new candidate solution, which now conflicts at* $(v, 2)$*. We repeat the process, introducing* $\mathtt{p}[v, 2]$ *and branching. Re-planning* a_2 *again, the shortest path is of length* 5*; the propagator attempts to update the lower bound of* c_2*, but fails as this violates its (assumed) upper bound. The solver generates a* nogood $\langle p[u, 1] \neq 2 \rangle \wedge \langle p[v, 2] \neq 2 \rangle \rightarrow \langle c_2 \geq 5 \rangle$ *recording this discovery. The solver then backtracks, and the nogood forces* $p[v, 2] = 2$*, which causes* a_1 *to be re-planned. We continue this process, eventually introducing* $\mathtt{p}[w, 2]$ *and deriving nogoods which cut-off all remaining leaves of the subtree in Figure 1(b), at which point the solver concludes there is no way for* a_1 *and* a_2 *to simultaneously achieve their minimum-cost solution, returning a core* $\{acc_1 \leq 5 \wedge \langle c_2 \leq 5 \rangle\}$ *expressing the constraint* $\langle c_1 \leq 5 \rangle \wedge \langle c_2 \leq 4 \rangle \rightarrow false$*.*

Using this core we relax bounds on a_1 *and* a_2 *to be* 6 *and* 5 *respectively, but allowing the combined cost to increase by* 1*.*

Trying again, the solver starts following its most recent solution. It chooses $\mathtt{p}[u, 1] = 2$ *and* $\mathtt{p}[x, 3] = 2$*. The pathfinder for* a_1 *runs, discovering the shortest path for* a_1 *is now length* 6*. With the current assumptions, this forces the maximum cost for* a_2 *to* 4*. The solver now returns a minimum-cost plan with no conflicts between* a_1 *and* a_2*, but a conflict remains between* a_3 *and* a_4*. We repeat the same process of introducing variables and re-solving, until the solver derives a new core* $\{\langle c_3 \leq 4 \rangle, \langle c_4 \leq 4 \rangle\}$*. We again relax bounds on* a_3 *and* a_4*, re-solve under the new assumptions. The solver returns a new plan which is optimal with respect to the constraints so far, and which has no conflicts – this is our final, cost-optimal plan.*

Here we see some key differences from 'traditional' CBS: exploration proceeds depth-first, observed constraints remain persistent across branches of the search tree, and – most critically – so is information we learn about cost and feasibility of *partial* plans. This allows us to avoid the behavior we observe in Example 1: each time a subproblem fails because of the bound on c_3, conflict analysis discovers that it is dependent only on constraints involving a_3 and a_4, so does not need to explore other ways of resolving (a_1, a_2) conflicts.

Explaining path costs

In order to make use of LCG solving, we must also equip our `path` propagators with explanations: whenever we conclude $c_i \geq k$ because the shortest path for agent i must be at least length k given the values of the $\mathtt{p}$ variables, we must also be able to identify a subset of obstacles (constraints on the $\mathtt{p}$ variables) sufficient to force the bound change. A simple strategy, which is certainly sound, is to collect the set of obstacles, O, which are currently enforced for agent a_i (as a set of atomic constraints), i.e. $O = \{\langle \mathtt{p}[l,t] \neq i \rangle \mid \mathtt{p}[l,t] \neq i\}$.

However, this produces explanations with limited reusability. By modifying algorithms used for explanations on MDDs (Gange, Stuckey, and Szymanek 2011), we can identify *minimal* explanations. The intuition for the algorithm is reasonably simple. Starting from the goal location, we sweep backwards, identifying *forbidden* (l, t) pairs which, if they were reachable from the start, would complete a path of cost less than k. Then, we sweep forward from the start, discarding any obstacles which do not complete a forbidden path.

An algorithm for collecting explanations is give in Figure 3. G is the input graph, s and g the start and goal locations, O the set of active obstacles for the agent, and k the bound to be explained. EXPLAIN-LB(G, i, s, g, O, k) returns a minimal subset of constraints O (though not necessarily minimum-size) which require agent i to take a path of length at least k. The algorithms make use of a heuristic $H(l_1, l_2)$ which returns a lower bound on the distance from location l_1 to location l_2. This can be precomputed for the map G for MAPF problems, or a simple Manhattan distance can be used.

Example 6 *Recall the problem illustrated in Figure 1. Assume a_1 has been forbidden entering* all *of $\{u, v, w, x\}$ at its optimal time, blocking all paths of length* 5.

Figure 4 illustrates the computation of EXPLAIN-LB. *During* MARK-FORBIDDEN, *we explore backwards from g_1; in this case, $\{v, w, x\}$ all admit paths to g_1 which do not pass through any other obstacles. u is not marked as forbidden, because any sufficiently short path through u must later pass through either v or w.*

Then, COLLECT-SUFFICIENT *explores forward from s_1, collecting a minimal set of obstacles. Here u and v are both reachable and forbidden, so must remain part of the explanation. x is never reached during the* COLLECT-SUFFICIENT, *because all sufficiently short paths through x are eliminated by including u and v. We thus return the explanation $p[v, 2] \neq 1 \wedge p[w, 2] \neq 1 \rightarrow c_1 \geq 6$.*

```
1:  function EXPLAIN-LB($G, i, s, g, O, k$)
2:      $F \leftarrow$ MARK-FORBIDDEN($G, i, s, g, O, k$)
3:      $E \leftarrow$ COLLECT-SUFFICIENT($G, i, s, g, O, k, F$)
4:      return $E$
5:  function MARK-FORBIDDEN($G, i, s, g, O, k$)
6:      $F \leftarrow \emptyset$
7:      $Q \leftarrow \{(g, k-1)\}$
8:      while $Q \neq \emptyset$ do
9:          $(l, t) \leftarrow$ POP($Q$)
10:         if $\langle \mathtt{p}[l,t] \neq i \rangle \in O$ then
11:             $F \leftarrow F \cup \{\langle \mathtt{p}[l,t] \neq i \rangle\}$
12:         else
13:             for $l' \in \{l\} \cup$ ADJACENT($G, l$) do
14:                 if $H(s, l) \leq t - 1 \wedge (l', t-1) \notin Q$ then
15:                     $Q \leftarrow Q \cup \{(l', t-1)\}$
16:     return $F$
17: function COLLECT-SUFFICIENT($G, i, s, g, O, k, F$)
18:     $E \leftarrow \emptyset$
19:     $Q \leftarrow \{(s, 0)\}$
20:     while $Q \neq \emptyset$ do
21:         $(l, t) \leftarrow$ POP($Q$)
22:         if $\langle \mathtt{p}[l,t] \neq i \rangle \in F$ then
23:             $E \leftarrow E \cup \{\langle \mathtt{p}[l,t] \neq i \rangle\}$
24:         else
25:             for $l' \in \{l\} \cup$ ADJACENT($G, l$) do
26:                 $t_E \leftarrow t + 1 + H(l', g)$
27:                 if $t_E < k \wedge (l', t+1) \notin Q$ then
28:                     $Q \leftarrow Q \cup \{(l', t+1)\}$
```

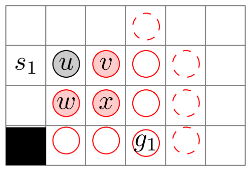
Figure 3: Explaining lower bound changes for a single agent.

Failure-guided search with unsatisfiable cores

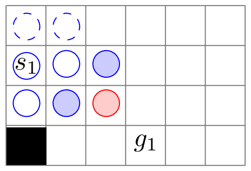
Core-guided search makes assumptions that a set of literals are true, and then either finds a solution where all literals are true or returns a subset of the assumptions which cannot all be simultaneously true. In our case, the assumptions are about the path length of the agents as well as some artificial terms added to implement OLL core-guided search (Andres et al. 2012). The algorithm is shown in Figure 5. Given a core of size m we first add a nogood defining the core to the solver. We introduce a variable t taking values in the range $1..m$ to bound the number of literals true in the new nogood. We can set t's lower bound to 1, since we know at least 1 literal must be true. We relax all the bounds on the variables in the core by 1, allowing them to take larger values. We then add an assumption that the upper bound of t is 1, which requires at most one literal to take a larger value.

Example 7 *Recall our problem from Figure 1. Each agent has a shortest path of cost 4, except for agent 1 with a shortest path of 5. We initially try solving with the assumptions $\langle c_1 \leq 5 \rangle \wedge \langle c_2 \leq 4 \rangle \wedge \langle c_3 \leq 4 \rangle \wedge \langle c_4 \leq 4 \rangle$, that is each agent should takes its shortest path distance.*

After resolving some conflicts (as described above), the solver finds there is no solution, with a nogood $\langle c_1 \leq 5 \rangle \wedge$



(a) MARK-FORBIDDEN



(b) COLLECT-SUFFICIENT

Figure 4: Explaining $c_i \geq 6$. Filled nodes are obstacles, red nodes are explored by MARK-FORBIDDEN, blue nodes explored by COLLECT-SUFFICIENT. Dashed nodes were pruned due to H.

```
1: function PROCESS-CORE($s, \{at_1, \ldots, at_m\}, lb, L$)
2:     where $at_i \equiv \langle c_i \leq L[c_i] \rangle$
3:     $s \leftarrow$ CONSTRAIN($s, \vee_{i=1}^{m} \neg \langle c_i \leq L[c_i] \rangle$)
4:     $s, t \leftarrow$ NEW-VAR($s, 1..m$)        ▷ A new penalty term.
5:     for $i \in 1..m$ do
6:         $L[c_i] \leftarrow L[c_i] + 1$    ▷ Relax bounds on the core
7:         $s \leftarrow$ CONSTRAIN($s, t \geq \sum_{i \in 1}^{m} (c_i \geq L[c_i])$)
8:     $L[t] \leftarrow 1$          ▷ But restrict the new penalty term.
9:     return $lb + 1, L, s$
```

Figure 5: Processing a core produced by the solver. At least one member of the core must take a sub-optimal value. We relax *all* members of the core, but introduce a fresh penalty term t to represent the number of violations. Here s is the LCG solver state, and L the current cost bounds.

$\langle c_2 \leq 4 \rangle \rightarrow$ *false – indicating a cardinal conflict between agents* a_1 *and* a_2. *We then relax the bounds on both agents, but we introduce a new variable* $t_1 = \langle c_1 \geq 6 \rangle + \langle c_2 \geq 5 \rangle$ *which counts the number of agents in* $\{a_1, a_2\}$ *that take a value greater than their lower bound. Since* t_1 *is constrained to be at least 1, this forces at least one agent to take a longer path, i.e. enforcing the nogood* $\langle c_1 \geq 6 \rangle \vee \langle c_2 \geq 5 \rangle$ *representing the core.*

The next iteration solves under the assumptions $\langle c_1 \leq 6 \rangle \wedge \langle c_2 \leq 5 \rangle \wedge \langle t_1 \leq 1 \rangle \wedge \langle c_3 \leq 4 \rangle \wedge \langle c_4 \leq 4 \rangle$. *The assumption on* t_1 *ensures that at most one of agents 1 and 2 can take a longer path than its shortest path.*

Feasible paths for a_1 *and* a_2 *are quickly found, paths for* a_3 *and* a_4 *are found to be in conflict. The same search is performed, and an unsat core* $\langle c_3 \leq 4 \rangle \wedge \langle c_4 \leq 4 \rangle \rightarrow$ *false is discovered. We process this core, relaxing* c_3 *and* c_4*, and introduce new term* t_2 *to count how many agents in* $\{a_3, a_4\}$ *take a longer path, re-solving with assumptions* $\langle c_1 \leq 5 \rangle \wedge \langle c_2 \leq 5 \rangle \wedge \langle t_1 \leq 1 \rangle \wedge \langle c_3 \leq 5 \rangle \wedge \langle c_4 \leq 5 \rangle \wedge \langle t_2 \leq 1 \rangle$. *This iteration succeeds, finding a feasible (thus optimal) plan.*

Note that in later iterations the previously introduces t variables will take part in unsatisfiable cores, and their violations will be counted by newly introduced t variables. This implements the OLL approach to unsatisfiable core optimization, see (Andres et al. 2012) for more details.

Experimental Evaluation

We have developed a prototype of our proposed approach (available at https://bitbucket.org/gkgange/lazycbs), taking the low-level pathfinder of a CBS-based MAPF solver and embedding it as a propagator in `geas` (https://bitbucket.org/gkgange/geas), a lazy clause generation-based constraint programming solver.

For evaluation, we use 4 maps (two small grids, and two game maps) with varying numbers of agents. For each map and instance size, we generated 50 instances allocating random (disjoint) start and goal locations to each agent. Experiments are conducted on an Intel Core i7-7820HQ with 32Gb RAM, running Lubuntu 17.10, and all experiments were run with a 5 minute time limit. To evaluate the effect of replacing the high-level solver with Lazy CBS, we compare with an implementation of CBS (Sharon et al. 2015) running the same low-level pathfinder as Lazy CBS. We also compare with the current state of the art implementations of CBSH (Felner et al. 2018), and ECBS (Barer et al. 2014) using a 1% sub-optimality threshold.

Figure 6 gives results on a 4-connected 20 × 20 grid with no obstacles (`20x20`) and with 10% blocked cells (`10obs-20x20`). We considered examples with 20, 30, 40, 50, and 60 agents.

We show for each class of map both: the *success rate* for each algorithm, that is how many of the 50 instances were solved to optimality in the time given; and a *cactus plot* of the solve time, showing the total number of instances solved by each algorithm by a given time limit. In terms of success rates, Lazy CBS is considerably more robust, and scales better, than CBS. If we consider solution time, we can see that CBS solves easy instances – requiring little search – faster, but performance degrades very rapidly. Whereas Lazy CBS suffers on easy instances (due to additional pathfinder calls during depth-first exploration), but displays much more stable performance. CBSH is much more robust than CBS on these maps due to its use of cardinal conflict reasoning, but as the number of agents grow its success rate dramatically drops. The cactus plot comparison shows that CBSH also has significant overhead above CBS for easy instances, its always bettered by Lazy CBS beyond the easiest instances.

For ECBS, performance is highly dependent on the sub-optimality bound. Where some feasible solution exists within the initial sub-optimality bound, ECBS typically finds a solution very quickly. But where the sub-optimality bound is tight enough that the low-level pathfinder cannot bypass all conflicts, we see Lazy CBS displaying higher success rates than the suboptimal ECBS.

Figure 7 shows results on two standard benchmark grids, `den520d` and `lak503d`, derived from the game Dragon Age: Origins (Sturtevant 2012). On these maps, each pathfinder call is much more expensive, so the overhead of depth-first exploration is higher. But though the break-even point is later, we see the same pattern emerging: Lazy CBS suffers a modest overhead on problems with very little search, but displays much more robust performance as problems become more difficult. The like to like comparison (in terms of low-level search) shows that Lazy CBS is much more effective than CBS. The advantage over CBSH in

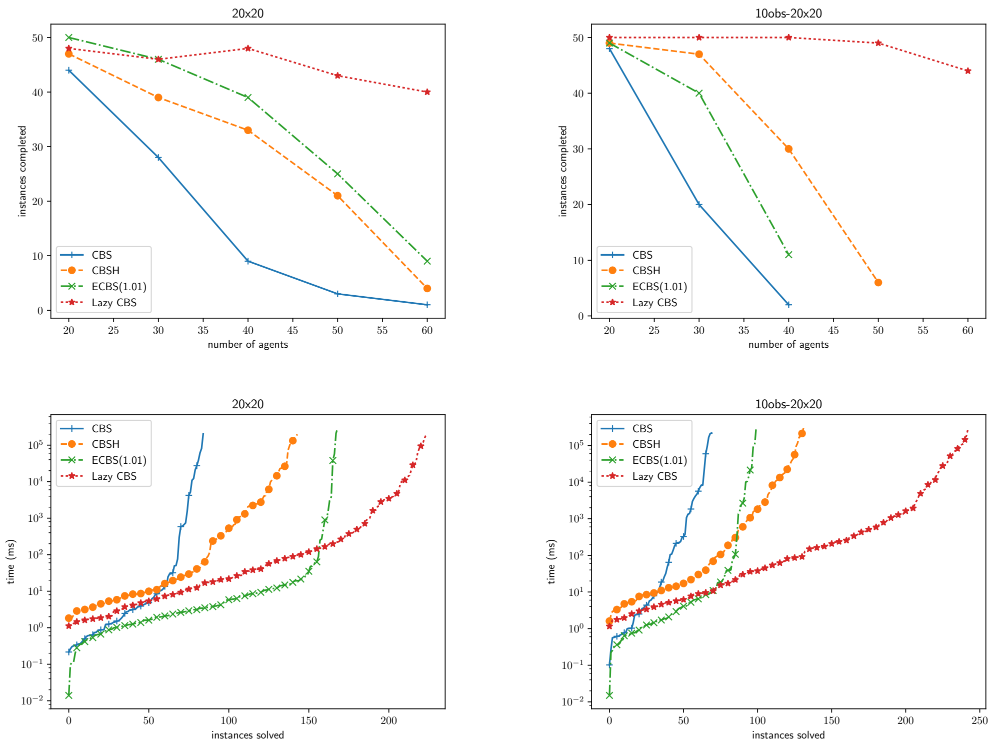


Figure 6: Results on an open 20×20 grid, and a 20×20 grid with 10% obstacles.

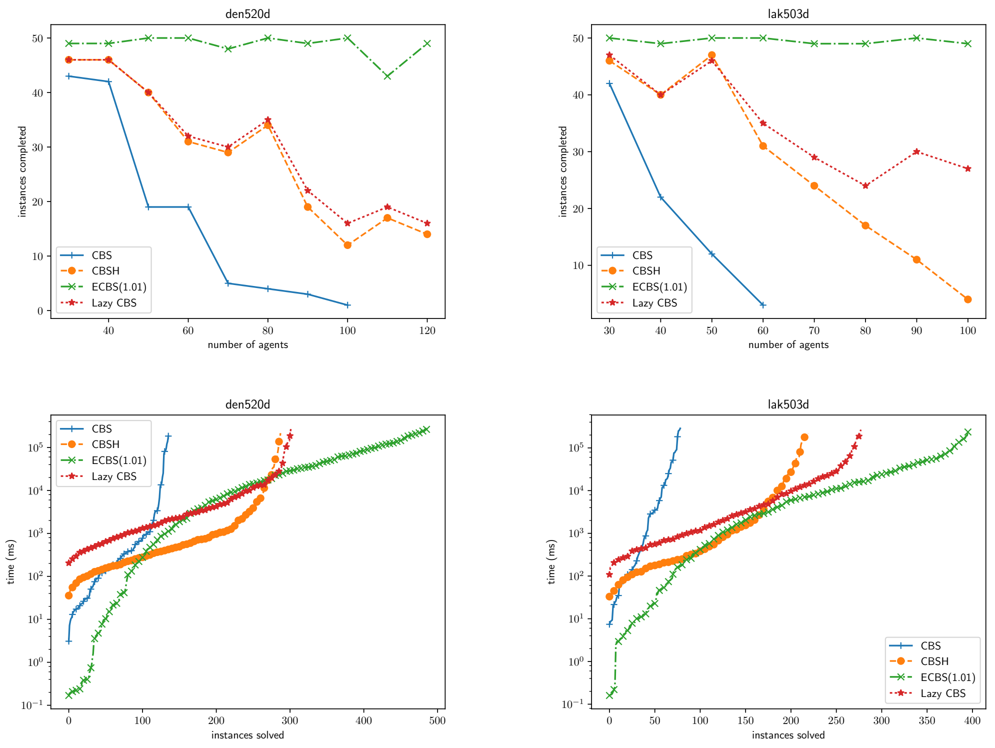


Figure 7: Results on the game maps `den520d` and `lak503d`.

terms of success rate is less apparent here, although clearly better as the number agents grow on `lak503d`. The cactus plot shows that for the `den520d` Lazy CBS only outpaces CBSH for the last, hardest instances. As the game maps are much larger (and paths therefore longer), we see success rates of ECBS remain stable here, as even 1% provides a enough flexibility in the low-level paths to bypass conflicts.

Clearly, Lazy CBS provides dramatically better performance compared to CBS. Indeed, despite having no MAPF-specific reasoning in the LCG solver, Lazy CBS is already competitive with CBSH on all problems. And for small, highly contested maps, Lazy CBS is much more robust.

If we replace CBS with CBSH we will be able to make the much better decisions for the order of handling conflicts which is made possible by using MDDs to represent all possible shortest paths for each agent. This is one of the principal advantages of CBSH over CBS.

Conclusion and further work

We have presented Lazy CBS, a new approach to optimal multi-agent pathfinding. Experimental results demonstrate that this approach is dramatically more robust than existing conflict-based search approaches, especially in problems with many agents and high contention.

Though performance is already quite robust, there is room to improve the low-level planner to cooperate more effectively with a Lazy CBS approach – particularly exploiting the depth-first exploration by re-planning incrementally. Extending the solver to allow dynamic constraint addition at a solution, to avoid restarting the solve every time we find a solution, should also improve the lazy approach.

Acknowledgements

This work was supported by the Australian Research Council through awards DE160100568 and DE160100007. We would like to thank Jiaoyang Li for making available her CBSH implementation and experimental data, and Liron Cohen for making available his ECBS implementation and single-agent pathfinder.

References

Andres, B.; Kaufmann, B.; Matheis, O.; and Schaub, T. 2012. Unsatisfiability-based optimization in Clasp. In *Technical Communications of the 28th International Conference on Logic Programming*, 211–221. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Barer, M.; Sharon, G.; Stern, R.; and Felner, A. 2014. Suboptimal variants of the conflict-based search algorithm for the multi-agent pathfinding problem. In *Proceedings of the 21st European Conference on Artificial Intelligence*, 961–962. IOS Press.

Davies, J., and Bacchus, F. 2011. Solving MAXSAT by solving a sequence of simpler SAT instances. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming*, 225–239. Springer.

Davies, J., and Bacchus, F. 2013. Exploiting the power of MIP solvers in MAXSAT. In *Proceedings of the 16th International Conference on Theory and Applications of Satisfiability Testing*, 166–181. Springer Berlin Heidelberg.

Demirovíc, E.; Chu, G.; and Stuckey, P. J. 2018. Solution-based phase saving and large neighbourhood search. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming*, 99–108.

Eén, N., and Sörensen, N. 2003. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.* 89(4):543–560.

Felner, A.; Stern, R.; Shimony, S. E.; Boyarski, E.; Goldenberg, M.; Sharon, G.; Sturtevant, N. R.; Wagner, G.; and Surynek, P. 2017. Search-based optimal solvers for the multi-agent pathfinding problem: Summary and challenges. In *SoCS*, 29–37.

Felner, A.; Li, J.; Boyarski, E.; Ma, H.; Cohen, L.; Kumar, T. K. S.; and Koenig, S. 2018. Adding heuristics to conflict-based search for multi-agent path finding. In *Proceedings of the 28th International Conference on Automated Planning and Scheduling*, 83–87. AAAI Press.

Gange, G.; Stuckey, P. J.; and Szymanek, R. 2011. MDD propagators with explanation. *Constraints* 16(4):407–429.

Morris, R.; Pasareanu, C.; Luckow, K.; Malik, W.; Ma, H.; Kumar, S.; and Koenig, S. 2016. Planning, scheduling and monitoring for airport surface operations. In *AAAI-16 Workshop on Planning for Hybrid Systems*.

Ohrimenko, O.; Stuckey, P.; and Codish, M. 2009. Propagation via lazy clause generation. *Constraints* 14(3):357–391.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2015. Conflict-based search for optimal multi-agent pathfinding. *Artif. Intell.* 219:40–66.

Silver, D. 2005. Cooperative Pathfinding. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment Conference (AIIDE)*, 117–122.

Sturtevant, N. R. 2012. Benchmarks for grid-based pathfinding. *IEEE Trans. Comput. Intellig. and AI in Games* 4(2):144–148.

Surynek, P. 2012. Towards optimal cooperative path planning in hard setups through satisfiability solving. In *Proceedings of the 12th Pacific Rim International Conference on Artificial Intelligence*, 564–576. Springer.

Veloso, M. M.; Biswas, J.; Coltin, B.; and Rosenthal, S. 2015. Cobots: Robust symbiotic autonomous mobile service robots. In *IJCAI*, 4423.

Wurman, P. R.; D'Andrea, R.; and Mountz, M. 2008. Coordinating hundreds of cooperative, autonomous vehicles in warehouses. *AI Magazine* 29(1):9–20.

Yu, J., and LaValle, S. M. 2013. Structure and intractability of optimal multi-robot path planning on graphs. In *AAAI Conference on Artificial Intelligence*, 1444–1449.