Proceedings of the Ninth International
Symposium on Combinatorial Search (SoCS 2016)

Recursive Constraint Manifold Subsearch for Multirobot Path Planning with Cooperative Tasks

Peter Karkus
NUS Graduate School for Integrative Sciences
and Engineering, School of Computing
National University of Singapore

Glenn Wagner, Howie Choset
Robotics Institute
Carnegie Mellon University

Abstract

The Cooperative Path Planning (CPP) problem seeks to determine a path for a group of robots which form temporary teams to perform tasks. The multi-scale effects of simultaneously coordinating many robots distributed across the workspace while also tightly coordinating the members of teams increases the difficulty of planning. Previous research produced the Constraint Manifold Subsearch (CMS) algorithm that can find minimal length paths to the CPP problem. However, CMS as currently formulated cannot handle more general cost functions, such as minimizing energy expenditure, and cannot handle task schedules that require multiple input teams to merge to form a set of multiple output teams. Furthermore, as CMS must couple planning for all interacting teams, it does not scale well to very large environments. In this paper, we rederive the CMS algorithm using a task graph to reason about inter-team dependencies, allowing the use of more general cost functions and task schedules. We then introduce the recursive CMS (rCMS) algorithm that exploits the reformulation to split the CPP into independent subproblems, significantly reducing computational complexity. Simulation studies show that rCMS can solve substantially larger problems than CMS.

Introduction

Many interesting problems, such as automated assembly or observation of multiple targets, involve tasks where robots must temporarily form cooperative teams. Handling the dynamic formation and dissolution of these robot teams requires solving two problems: assigning robots to tasks, and coordinating the motions of large numbers of robots. In this work, the assignment of robots to tasks and the order in which tasks must be executed are assumed to be provided *a priori*, and the focus will be on finding optimal or near-optimal, collision-free paths for large numbers of robots that dynamically form teams, which we term the *Cooperative Path Planning* problem.

This work is based on the Constraint Manifold Subsearch (CMS) algorithm (Wagner et al. 2015) for solving the CPP problem, which in turn is based on the M* algorithm (Wagner and Choset 2011) for solving the related multirobot path planning algorithm. M* finds optimal or near optimal paths for large numbers of individual robots from an initial configuration to a goal configuration by decoupling planning for separate robots when possible. Initially planning is conducted for each robot separately in its individual configuration space, then M* (locally) merges the configuration spaces of multiple robots only after finding a potential collision. However, M* cannot handle the CPP problem, which requires robots to form tightly coupled teams. CMS extends M* to solve the CPP problem by temporarily merging the robots in a team into a single meta-agent whose configuration space is the *constraint manifold* of the task, *i.e.* the subspace of the team configuration space that satisfies the constraints of the task.

CMS is complete and guaranteed to find minimal length paths, or can accept ϵ-*suboptimality* in return for greatly reduced planning time. However, CMS cannot minimize more general cost functions, such as energy expenditure, and cannot handle complex transitions where multiple teams each containing multiple robots merge to form a new set of two or more teams. Finally, CMS scales poorly in large environments, as CMS must couple planning for all robots for which CMS has found a potential collision, even if there are multiple, spatially separated collisions.

This paper reformulates the CMS algorithm by using a *task graph* to reason about dependencies between teams of robots. The reformulation allows for more general cost functions and for more complex transitions between teams. Furthermore, the reformulation makes identification of independent subproblems easier. This leads to the introduction of the recursive CMS (rCMS) algorithm, which reduces computational complexity by identifying and solving independent subproblems. Simulation studies demonstrate that rCMS can solve substantially larger problems than CMS.

Prior Work

Solving the CPP problem requires solving two qualitatively different problems: coordinating the simultaneous motion of many teams of robots (Wagner and Choset 2011) (Felner et al. 2012) (Sharon et al. 2012) (Standley 2010) and finding paths for the robots within a team that satisfy the constraints of the task being executed. While there has been a large amount of work in formation control (Belta and Kumar 2001) (Leonard and Fiorelli 2001) (Lewis and Tan 1997) (Spears and Gordon 1999) (Yang, Freeman, and

Lynch 2008) and cooperative manipulation (Alonso-Mora et al. 2015) (Desai and Kumar 1999) (Koga and Latombe 1994) (Rus, Donald, and Jennings 1995), little work has been done in the context of path planning with the dynamic formation and dissolution of teams. We review some of the notable exceptions which have inspired our work here.

(Ayanian and Kumar 2010), (Desaraju and How 2012), and (Bhattacharya, Likhachev, and Kumar 2010) developed CPP planners that could plan for systems where multiple teams form and persist for significant durations. Ayanian and Kumar (Ayanian and Kumar 2010) solved problems where robots must remain in close proximity to the other robots in its team by searching the prepares graph of controllers in a sequential composition framework (Burridge, Rizzi, and Koditschek 1999), and later extended the work to consider dynamic formation and dissolution of teams (Ayanian 2011), but this algorithm did not scale well to larger numbers of robots or consider the dependencies between teams introduced by successive teams sharing robots. Desaraju and How (Desaraju and How 2012) and Bhattacharya, Likhachev and Kumar (Bhattacharya, Likhachev, and Kumar 2010) both developed CPP algorithms that operate incrementally by adjusting the path of one robot at a time to better match the constraints imposed by its tasks. Desaraju and How (Desaraju and How 2012) developed DM-RRT, a decentralized algorithm where a single robot was allowed to replan its path to better match the task constraints. DM-RRT scales well with increasing numbers of robots, but guarantees neither completeness nor optimality. Bhattacharya, Likhachev and Kumar (Bhattacharya, Likhachev, and Kumar 2010) repeatedly replanned paths for individual robots while gradually increasing the cost of violating task constraints. The resulting approach can be shown to eventually converge to the optimal path, but does not scale well with increasing numbers of robots.

Problem Definition

The objective of the CPP problem is to find a minimal cost, collision-free path for teams of robots that complete a set of m cooperative tasks. The configuration space of each robot $r^i, i \in \{1, 2, \dots n\}$ is represented by a weighted *configuration graph* $G^i_{\text{conf}} = \{V^i_{\text{conf}}, E^i_{\text{conf}}\}$, where vertices represent the configuration of r^i and edges represent valid actions. The weight of each edge is the non-negative cost associated with each action. The *joint configuration space* of the system as a whole is represented by the *joint configuration graph* $G_{\text{conf}} = \prod_i G^i_{\text{conf}}$ which is the direct product of the individual configuration graphs. The cost of an edge in the joint configuration graph is the sum of the costs of the underlying edges in the individual configuration graphs. The configuration graph of a team of robots is the direct product of the configuration graphs of the constituent robots. An optimal solution to the CPP problem is a path that minimizes the sum of the costs of the edges in the path while completing all tasks and avoiding collisions between robots.

Each task is represented by a tuple $(\tau^j, v^j_s, v^j_f, \mathfrak{C}^j), j \in \{1, 2, \dots, m\}$ where τ^j is the team of robots assigned to perform the task, v^j_s is the vertex in the configuration graph of τ^j at which task execution can start, v^j_f is the goal vertex in the configuration graph of τ^j at which the task can be completed, and $\mathfrak{C}^j$ is a set of *task constraints* that the robots in τ^j must satisfy during task execution. Examples of task constraints include robots carrying a rigid body needing to stay a constant distant apart, or robots observing a target needing to stay on opposite sides to keep the full surface of the target in view. For simplicity, we assume that every robot is always part of a (potentially single robot) team.

In some cases multiple teams must cooperate to complete their tasks. For instance, multiple small teams may be tasked to pick up a single large object, forming a new, large team to carry the object to its destination. Clearly none of the small teams lift the object until all are at their goal configurations. To model such situations, a team is required to take an explicit *transition action* while at its goal configuration to complete its task. Each transition action has a set of *input teams*, all of which must take the transition action simultaneously to complete their tasks, and a set of *output teams* that form from the robots constituting the input teams when the transition action is performed.

The order in which teams perform their tasks and the input and output teams for each transition action are encoded in a directed, acyclic *task graph*, a standard tool for reasoning about task ordering (Graham et al. 1979) (Hu 1961). Each vertex corresponds to either a team or a transition action. Edges connect each input team to its corresponding transition action, and connect a transition action to its output teams.

This paper makes two simplifying assumptions. Teams are assumed to be able to wait indefinitely at their goal vertices. Furthermore, teams incurs zero cost for waiting at their goal vertices if they cannot take the transition action due to the other input teams not being at their goal vertices or if the team does not have a successor in the task graph (i.e. is a terminal team).

Constraint Manifold Subsearch

CMS (Wagner et al. 2015) is based on a multirobot path planning algorithm called M* (Wagner and Choset 2011). M* seeks to efficiently find optimal or near optimal paths for multirobot systems by initially planning for each robot separately, then only considering alternate paths for robots that are directly involved in conflicts with other robots. CMS extends M* to the CPP problem by adjusting the search space so every team is explicitly restricted to the constraint manifold[1] of its task, the subspace of the configuration space of the team that satisfies the task constraints. A team is restricted to its constraint manifold by temporarily merging its constituent robots into a single meta-agent whose configuration space is the constraint manifold of the task.

CMS represents the constraint manifold $\mathcal{M}^i$ of team τ^i with the weighted, directed, *manifold graph* $G^i = \{V^i, E^i\}$. Vertices $v^i \in V^i$ represent valid team configurations, while edges in E^i represent actions of the team. The cost of an

[1] Constraint manifold is a term of convenience: CMS can solve problems where the task constraints are satisfied on a subspace that is not a manifold.

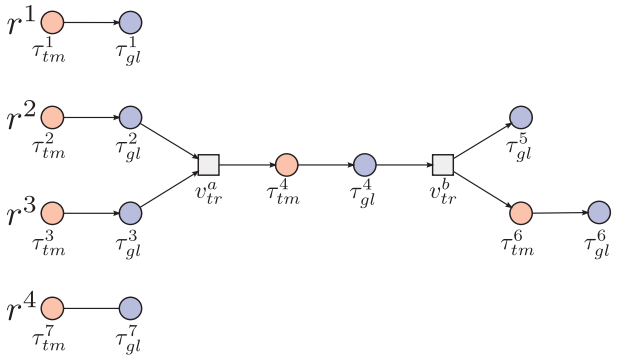

Figure 1: Example of a task graph. Robot r^1 is the sole robot assigned to team τ^1; r^2 is assigned to teams τ^2, τ^4, and τ^5; r^3 is assigned to τ^3, τ^4, and τ^6; and r^4 is the only robot assigned to τ^7. Note that $v^5_s = v^5_f$, so τ^5 is represented by a single vertex in V_{gl} (blue), and has no corresponding vertex in V_{tm} (red). The transition vertices are the gray squares.

edge is the cost associated with the team action. v^i_s and v^i_f are henceforth treated as vertices in G^i. Different manifold graphs may discretize the configuration space of the same robot differently; an independent robot may move on a rectangular grid, but move on circular arcs when part of a formation to permit rotation.

CMS finds a solution to the CPP problem by exploring a graph that is the product of the manifold graphs of the teams. As in the case of M*, the resulting graph is far too large to explore directly. CMS first plans for each team separately. When CMS finds a conflict between teams in the resulting path, it considers alternate paths for only those teams that could influence the path taken by the conflicting teams. As formulated in (Wagner et al. 2015), CMS determines if the path of one team depends upon that of another team by checking if their constituent robots overlap. This criteria is sufficient as long as the cost function being minimized penalizes time spent away from the goal configuration of each team, and every transition between teams consists of multiple teams merging to form a single new team or a single team splitting into multiple smaller teams.

In this section, we reformulate CMS to use the task graph to reason about the dependencies between teams. The revised algorithm is simpler, supports more general cost functions that are the sum of the costs of the individual teams, and supports arbitrary transitions between teams. We start by describing how the task graph is used to reason about dependencies between teams, followed by an algorithmic description of CMS.

The Task Graph

If team τ^i is involved in a conflict with another team then there are three ways that the path of τ^i can be altered to avoid the collision

1. τ^i can take a different path from v^i_s to v^i_f
2. The teams that precede τ^i can alter their paths so that τ^i forms earlier or later, altering the position of τ^i relative to other moving teams
3. The other input teams for the transition action that completes τ^i can take shorter, but potentially more expensive, paths to reach their goal vertices earlier, allowing τ^i to take the transition action before the conflict would occur.

Note that case 3 is only relevant if τ^i had reached its goal vertex at least once prior to the conflict, or it would be unable to have taken the transition action prior to the conflict regardless of the paths of the other input teams. These rules dictate the teams for which CMS must consider alternate paths to optimally resolve a conflict involving τ^i. We note that the previous formulation of CMS did not consider case 3, which limited it to cost functions that minimized time to complete tasks, that naturally cause teams to reach their goal configurations as rapidly as possible.

CMS computes the teams that can resolve a conflict involving τ^i using a modified task graph. First note that case 3 means that the team dependencies change once τ^i reaches its goal vertex for the first time[2]. Therefore, τ^i is labeled τ^i_{tm} before it reaches its goal vertex, and τ^i_{gl} after it has reached its goal vertex. To simplify later discussion, τ^i is used to denote both τ^i_{tm} and τ^i_{gl} in cases where the distinction is not relevant. CMS then defines the task graph as a directed tripartite graph $G_{\text{task}} = \{V_{\text{tm}}, V_{\text{gl}}, V_{\text{tr}}, E_{\text{task}}\}$. The vertices $\tau^i_{tm} \in V_{\text{tm}}$ and $\tau^i_{gl} \in V_{\text{gl}}$ are teams before and after reaching their goal vertices, respectively. The transition vertices $v_{tr} \in V_{\text{tr}}$ represent transition actions. In the special case where $v^i_s = v^i_f$, for instance because τ^i is a single robot that must wait for another team to arrive to form its next team, the task graph contains τ^i_{gl} but not τ^i_{tm}. Let $v_{\text{task}} \in V_{\text{tm}} \cup V_{\text{gl}} \cup V_{\text{tr}}$ denote an arbitrary vertex in the task graph. Figure 1 gives an example of the task graph.

The task graph defines a partial ordering that describes the sequence in which teams must execute their tasks, wherein a vertex $v^i_{\text{task}} \in G_{\text{task}}$ is *dominated* by $v^j_{\text{task}} \in G_{\text{task}}$, denoted $v^i_{\text{task}} \leq v^j_{\text{task}}$, if there is a path in G_{task} from v^i_{task} to v^j_{task} or $v^i_{\text{task}} = v^j_{\text{task}}$. Similarly, $v^i_{\text{task}} < v^j_{\text{task}}$ if $v^i_{\text{task}} \leq v^j_{\text{task}} \wedge v^i_{\text{task}} \neq v^j_{\text{task}}$. Two vertices are incomparable if there is no path between them in G_{task}. Finally, a team or transition vertex is dominated by a set of task graph vertices if it is dominated by at least one element of the set.

Case 1 states that CMS must consider alternate paths for teams τ^j that must finish their tasks before τ^i can start, which corresponds to $\tau^j < \tau^i$. If case 3 is relevant, then $\tau^i = \tau^i_{gl} \in V_{\text{gl}}$, and its successor is a transition vertex which dominates all of its input teams. CMS can thus use the successor of τ^i to determine the relevant teams, as if $\tau^i \in V_{\text{tm}}$ its only successor is τ^i_{gl} whose only direct predecessor is τ^i. We therefore conclude that if τ^i is involved in a collision, then CMS must consider alternate paths for all teams τ^j such that $\tau^j < \mathit{successor}\left(\tau^i\right) \vee \tau^j = \tau^i$, where $\mathit{successor}\left(\tau^i\right)$ is the successor of τ^i in the task vertex. The $\tau^j = \tau^i$ condition is necessary for terminal teams that have no successors in the task graph, such as τ^7_{gl} in Figure 1.

[2] τ^i may temporarily move away from v^i_f to avoid other conflicts.

Algorithm 1 FindPath

```
1: function FINDPATH(v)
2:   open_list ← {v}
3:   while open_list ≠ ∅ do
4:     v_k ← open_list.pop_best()
5:     if v_k.closed then
6:       continue
7:     if ISGOAL(v_k) then
8:       return follow back_ptr to find optimal path to v_k
9:     EXPAND(v_k, open_list)
10:  return No Solution

11: function ISGOAL(v)
    Returns true if all teams in v are at their goal configuration and have no successors in the task graph
```

Applying the above reasoning to conflicts involving multiple teams gives the following lemma

Lemma 1. *A conflict between a set of teams C can be optimally resolved by considering alternate paths for all τ^i such that $\tau^i < successor\,(C) \vee \tau^i \in C$, where $successor\,(C)$ is the set of successors of teams in C.*

Algorithmic description of CMS

CMS functions by alternating between running A* search on a low-dimensional *search graph* and expanding the search graph to generate alternate paths around potential conflicts found by the A* search. The search graph is initially constructed using the *individual policies* of the teams, that describe the optimal path for each team if no robot-robot collisions could occur. Each vertex v_k in the search graph maintains a *conflict set* C_k that tracks the teams that potentially conflict at successors of the vertex, and a *coupled set* Γ_k of teams that might be able to avert the conflicts by changing their action at v_k. The coupled set is then used to grow the search graph to provide alternate paths around conflicts.

The search graph explored by CMS is a subgraph of the directed *task augmented joint configuration graph* G, which tracks the teams that are actively executing their tasks at a given point in the plan and the configuration of the active teams. Each vertex in G is a set of ordered pairs $\left(\tau^i, v^i\right)$, where τ^i is a team that is currently executing its task and v^i is a vertex in G^i describing the team's configuration. Let the *active teams* $\mathcal{T}_k^{\text{act}}$ denote the set of teams performing their task at v_k. Edges in G correspond to motion of the teams and transitions between teams.

CMS begins by computing a separate individual policy $\phi^i : V^i \mapsto V^i$ for each team. The individual policy for τ^i returns the next step along the optimal path from any vertex in G^i to v_f^i if no other teams were present. The minimal cost for τ^i to reach its goal from any given vertex in G^i can also be computed from the individual policy.

After the individual policies are computed, CMS enters a standard A* search loop (Algorithm 1). The vertices v_k in the open list are sorted by their *f-value*, $f = g + h$, which is the sum of a *g-value*, the current upper bound on the cost of the optimal path from the initial configuration to v_k, and a *h-value*, the heuristic cost to go. The heuristic used by CMS is the sum of the costs of the active teams and all subsequent teams (those not dominated by an active team) completing their tasks by following their individual policies. The resulting heuristic is admissible and monotonic.

Algorithm 2 Expand

```
1: function EXPAND(v_k, open_list)
2:   v_k.closed ← True
3:   neighbors ← GETNEIGHBORS(v)
4:   for v_ℓ ∈ neighbors do
5:     C = COLLISION(v_k, v_ℓ)
6:     BACKPROPAGATE(v_k, C, open_list)
7:     if C ≠ ∅ then
8:       continue
9:     v_ℓ.back_prop_set.add(v_k)
10:    BACKPROPAGATE(v_k, C_ℓ, open_list)
11:    if v_k.g + EDGECOST(v_k, v_ℓ) ≥ v_ℓ.g ∨ v_ℓ.closed then
12:      continue
13:    v_ℓ.g ← v_k.g + EDGECOST(v_k, v_ℓ)
14:    // used to reconstruct optimal path
15:    v_ℓ.back_ptr ← v_k
16:    open_list.insert(v_ℓ)

17: function COLLISION(v_k, v_ℓ)
    Returns the set of teams that collide when the system traverses the edge from v_k to v_ℓ

18: function BACKPROPAGATE(v_k, C, open_list)
19:   // Propagate if C changes the coupled set
20:   if COUPLEDSET(C) ⊄ COUPLEDSET(C_k) then
21:     C_k ← C_k ∪ C
22:     // Mark v_k as open to explore new neighbors
23:     v_k.closed ← False
24:     open_list.insert(v_k)
25:     for v_ℓ ∈ v_k.back_prop_set do
26:       BACKPROPAGATE(v_ℓ, C_k, open_list)
```

CMS calls the EXPAND function (Algorithm 2) to add neighbors of the expanded vertex to the open list, and it also handles detection of conflicts and updating the conflict and coupled sets when conflicts are detected. When a vertex v_k is expanded, CMS considers only a subset of all the neighbors of v_k in G termed the *limited neighbors*, which are determined by the coupled set of v_k. The limited neighbors are generated by the GETNEIGHBORS function (Algorithm 3), which is described in detail later in this section.

For each limited neighbor v_ℓ CMS checks if there are any teams that would collide while moving along the edge from v_k to v_ℓ. If so, then the colliding teams are added to the conflict set of v_k and all predecessors of v_k using the BACKPROPAGATE function. If updating the conflict set of v_k changes the coupled set of v_k (the computation of the coupled set from the conflict set is detailed later this section), then v_k has new limited neighbors. To explore these additional neighbors CMS adds v_k back to the open list, even if v_k had previously been marked as closed.

Algorithm 3 GetNeighbors

```
1: function GETNEIGHBORS(v_k)
2:   // Holds partial combinations of actions for teams
3:   neighbors ← {∅}
4:   // Generate possible actions one team at a time
5:   for (τ^i, v_k^i) ∈ v_k do
6:     // Temporary variable
7:     add_step ← ∅
8:     // Iterate combinations of actions for earlier teams
9:     for neib ∈ neighbors do
10:      if ∃τ^j ∈ neib, τ^i ≤ τ^j then
11:        // τ^i already assigned a transition action
12:        add_step.append(neib)
13:        continue
14:      add_step.append(neib ∪ GETPOLICY(v_k, τ^i, neib))
15:      // Consider all actions for teams in the coupled set
16:      if τ^i ∈ Γ_k then
17:        for v_ℓ^i ∈ OUTNEIGHBORS(v_k^i) do
18:          add_step.append(neib ∪ {(τ^i, v_ℓ^i)})
19:    neighbors ← add_step
20:  return neighbors
```

Whenever a (τ^i, v^i) pair is generated, test if $\tau^i \in V_{\text{tm}}$ and $v^i = v_f^i$. If so, replace τ^i with τ_{gl}^i

Algorithm 4 GetPolicy

```
1: // Generate action taken by τ^i when following its individual policy at v_k, including taking transition actions
2: function GETPOLICY(v_k, τ^i, neib)
3:   ret ← ∅
4:   // Take the transition action if possible, else follow individual policy
5:   if CANTRANSITION(τ^i, v_k, neib) then
6:     for τ^j ∈ OUTPUTTEAMS(τ^i) do
7:       ret ← ret ∪{(τ^j, v_s^j)}
8:   else
9:     ret ← {(τ^i, φ^i(v_k^i))}
     return ret

10: function CANTRANSITION(τ^i, v_k, neib)
    Returns True if every input team for the next transition action of τ^i is at its goal vertex in the configuration given by v_k, and no input team already has an action assigned in neib

11: function OUTPUTTEAMS(τ^i)
12:   Returns the output teams of the transition vertex for which τ^i is an input team.
```

If the edge connecting v_k to v_ℓ is collision free CMS marks v_k as a predecessor of v_ℓ by adding v_k to the backpropagation set (back_prop_set) of v_ℓ. Then BACKPROPAGATE is called to add the conflict set of v_ℓ to C_k. CMS then updates the g-value of v_ℓ and adds v_ℓ to the open list, if appropriate.

The GETNEIGHBORS function (Algorithm 3) returns the limited neighbors of a given vertex v_k, thus incrementally defining the search graph. Initially each team τ^i will take the action dictated by their individual policy. Teams at their goal vertices will take the transition action to finish their tasks if the other input teams are also at their goal vertices, or will stay in place if there are one or more input teams away from their goals. The result is a single limited neighbor. Considering only a single neighbor will generate a path that is almost certainly blocked by at least one conflict between teams. When CMS finds a conflict between teams at a successor of v_k, it adds the conflicting teams to the conflict set of v_k and marks v_k for reexpansion (Algorithm 2). If v_k has a non-empty conflict set when expanded CMS must consider all alternate paths that could prevent the conflict to ensure optimality. To this end, CMS computes a coupled set Γ_k containing all the teams that could alter the path taken by the teams in the conflict set (Lemma 1)

$$\Gamma_k = \left\{\tau^i \in \mathcal{T}_k^{\text{act}} | \tau^i < \mathit{successor}\left(C_k\right) \vee \tau^i \in C_k\right\}, \quad (1)$$

where $\mathit{successor}\left(C_k\right)$ is the set of the successors of the elements of C_k in the task graph. CMS then generates the limited neighbors by considering all possible combinations of actions for teams in the coupled set, while teams not in the coupled set continue to obey their individual policies (Algorithm 4).

As an example of how coupled sets are computed consider the case of the task graph given in Figure 1. If teams τ^1 and τ^2 were to collide before τ^1 and τ^2 reached their goal configuration, then the conflict set would be $\{\tau_{tm}^1, \tau_{tm}^2\}$. CMS may then backtrack in the search tree to consider alternate paths from a vertex v_ℓ with the active teams $\{\tau_{tm}^1, \tau_{tm}^2, \tau_{tm}^3\}$. The coupled set of v_ℓ would be the teams dominated by $\{\tau_{gl}^1, \tau_{gl}^2\}$, which are $\{\tau_{tm}^1, \tau_{tm}^2\}$. However, if CMS had found a potential collision between τ^1 and τ^2 after τ^2 had reached its goal configuration then the conflict set would be $\{\tau_{tm}^1, \tau_{gl}^2\}$. If CMS then backtracked to the same vertex v_ℓ, the coupled set would be the teams dominated by $\{\tau_{gl}^1, v_{tr}^a\}$, which are $\{\tau_{tm}^1, \tau_{tm}^2, \tau_{tm}^3\}$. τ^3 would be added to the coupled set because τ^2 might be able to avoid the collision with τ^1 by taking the team formation action earlier. τ^2 can only take the transition action when τ^3 is also at its goal configuration, so τ^3 might need to take a shorter but more expensive path to allow the transition action to occur earlier.

Note that a collision between two single robot teams could eventually result in coupled planning for all robots in the system at some predecessor vertex, especially if a single robot is part of multiple teams containing different robots. In such cases finding an optimal solution would be computationally very expensive. Inflating the heuristic function by a constant factor ϵ biases search towards the final state of the system (Pohl 1973), which provides a soft limit on how far back in the search CMS will look for an alternate path around collisions, limiting the effective size of the coupled set. However, inflated CMS is ϵ-suboptimal, which means that it may return a path that costs up to ϵ times the cost of the optimal path.

In practice, CMS uses Operator Decomposition (OD)

Algorithm 5 GetNeighborsRecursive

// This function is used to solve subproblems, where transition actions may have different output teams than in the full problem

function GETNEIGHBORSRECURSIVE(v_k)
  if NUMELEMENTS(Γ_k)$= 1 \wedge \Gamma_k^1 = \mathcal{T}_k^{\text{act}}$ **then**
    return GETNEIGHBORS(v_k)
  // Initialize a vertex with an empty coordinate
  $v_\ell = \emptyset$
  // Compose partial solutions for each subproblem
  for $C_k^i \in C_k$ **do**
    query_vertex = $\left\{ \left(\tau^i, v_k^i\right) | \tau^i \in \Gamma_k^i \right\}$
    $v_\ell \leftarrow v_\ell \cup$GETSTEP($C_k^i$, query_vertex)
  for $\left(\tau^i, v^i\right) \in v_k$ **do**
    if $\exists \tau^j \in v_\ell, \tau^i \leq \tau^j$ **then**
      continue
    v_ℓ.append(GETPOLICY(v_k, τ^i))

function GETSTEP(C, v)
Queries the subplanner defined by C to find a solution to the associated subproblem starting from v. Returns the coordinate of the first step in the solution.

Whenever a (τ^i, v^i) pair is generated, test if $\tau^i \in V_{\text{tm}}$ and $v^i = v_f^i$. If so, replace τ^i with τ_{gl}^i. If τ_{gl}^i is not in the current subproblem, note the action but do not add (τ_{gl}^i, v^i) to the limited neighbor.

(Standley 2010), a variant of A* tuned for multiagent path finding, instead of basic A*. The algorithm CMS remains the same conceptually, requiring only minor implementation changes.

Recursive CMS

CMS couples planning for all teams in the coupled set, even when there are multiple physically separated collisions. Consider the task graph given in Figure 1. If τ_{tm}^1 was found to conflict with τ_{tm}^4, and τ_{tm}^2 was found to conflict with τ_{tm}^3, then CMS would consider all possible actions for the set of teams $\{\tau_{tm}^1, \tau_{tm}^2, \tau_{tm}^3, \tau_{tm}^4\}$. If each team had 5 valid neighbors, the GETNEIGHBORS function would have to generate $5^4 = 625$ neighbors. However in this problem, a planner should be able to resolve the collision between τ_{tm}^1 and τ_{tm}^4 independently of the collision between τ_{tm}^2 and τ_{tm}^3, where the teams in each subproblem would only have $5^2 = 25$ possible combinations of actions.

Recursive Constraint Manifold Subsearch (rCMS) exploits disjoint subproblems using a similar approach to rM* (Wagner and Choset 2011). The conflict set is split into a set of *conflict set elements*, where each conflict set element contains a disjoint set of conflicting teams. When rCMS expands a vertex in the search graph, it generates a single limited neighbor for each vertex by recursively calling subplanners to compute a partial solution that resolves the conflicts contained in each conflict set element separately. This operation resembles temporarily merging the teams that can help resolve a given conflict into a single meta-agent, then computing a policy for that meta-agent. rCMS then combines the partial solutions to generate a single combinations of actions (neighbor) for the full system. The challenge in rCMS is to identify the proper subproblems to solve for each conflict.

The subproblems defined by the conflict set elements must satisfy three properties: they must contain disjoint sets of teams, they must contain all alternate paths for the teams capable of resolving the conflict, and when their solutions are composed into a solution for the full problem any remaining conflicts must not be dominated by any of the existing conflict set elements. The first property ensures that the subproblems specify a unique action for each team. The second property ensures that the resolution of each conflict will be optimal, and the third will ensure that rCMS will always make progress in finding a solution to the full problem.

The subproblem defined by a conflict set element C^i must contain all teams which can influence the resolution of said collision, as identified by Lemma 1. The subproblem is thus defined by the subgraph of the task graph induced by the *resolve set* $\mathcal{T}_{res}\left(C^i\right)$

$$\mathcal{T}_{res}\left(C^i\right) = \left\{ v_{\text{task}}^j | v_{\text{task}}^j < successor\left(C^i\right) \vee v_{\text{task}}^j \leq C^i \right\}. \tag{2}$$

We term the result the subproblem task graph. The subproblem is formed from the specification of the full problem by removing any team that is not in the subproblem task graph. Note that this may change the output teams of some transition action. In the case that a transition action in the subproblem task graph has no output teams, or τ_{tm}^i is in the subproblem task graph but τ_{gl}^i is not, then the transition action (respectively the action taking τ_{tm}^i to its goal vertex) remains part of the solution, but the output teams (resp. τ_{gl}^i) are not generated.

A subproblem is solved once rCMS finds a vertex which either has no active teams, or every active team is at its goal vertex and has no successor in the full task graph (i.e. is a terminal team). Such a vertex would occur after every team in the conflict set has taken its transition action. Because a solution to a subproblem is guaranteed to be collision free, this stopping criteria ensures that the path for the full problem formed by composing the solutions to the subproblems will not contain any conflicts dominated by an existing conflict set element. As a result, the composed path will either by collision free, and thus a valid solution to the full problem, or will contain a conflict that defines a larger subproblem, ensuring that rCMS will always make forward progress.

Finally, given a set of conflicts, rCMS must ensure that the subproblems called to resolve the conflicts are disjoint, to ensure that the path chosen for each team is unique. Therefore, rCMS merges conflict set elements, and thus their associated subproblems, if the coupled sets computed for the conflict set elements according to Equation 1 overlap. Consider the following example using the task graph in Figure 1. Assume that rCMS has found a conflict between τ_{tm}^1 and τ_{gl}^5, and a second conflict between τ_{gl}^6 and τ_{tm}^7, leading to the conflict set $\{\{\tau_{tm}^1, \tau_{gl}^5\}, \{\tau_{gl}^6, \tau_{tm}^7\}\}$. rCMS will then backtrack to consider alternate paths from vertices in

the search graph that precede the conflict. The subproblems for the two conflict set elements overlap, as τ^4_{tm} is dominated by both conflict set elements. However, rCMS only cares about the portion of the subproblem from the point where it queries the subproblem and the goal. Therefore, if rCMS expands a vertex v_k whose active teams are $\{\tau^1_{tm}, \tau^5_{gl}, \tau^6_{tm}, \tau^7_{tm}\}$ the two subproblems are effectively disjoint. However, if rCMS backtracks to a vertex v_ℓ whose active teams are $\{\tau^1_{tm}, \tau^4_{gl}, \tau^7_{tm}\}$ τ^4 would be in the subproblems associated with both conflict set elements, thus the conflict set elements must be merged at v_ℓ.

We will now provide an algorithmic description of rCMS. rCMS starts with a single subplanner which is assigned to solve the full problem. It calls the FINDPATH and EXPAND functions described in Algorithms 1 and 2 with minor modifications. The ISGOAL function is altered to return true if a vertex contains no active teams or all teams are at their goal configuration with no successor in the task graph of the full problem, not the subproblem. EXPAND (Algorithm 2) uses modified versions of the COLLISION and BACKPROPAGATE subroutines that account for the conflict set now being a set of conflict set elements. Specifically, COLLISION returns a set of conflict set elements corresponding to seperate conflicts between teams, and when BACKPROPAGATE adds C to C_k it takes the union of the two sets, then merges any conflict set elements whose coupled sets at v_k overlap. The primary difference is that EXPAND calls GETNEIGHBORSRECURSIVE (Algorithm 5) function to generate the limited neighbors, which contains most of the rCMS specific logic.

When generating the neighbors of v_k GETNEIGHBORSRECURSIVE first checks whether C_k contains a single conflict set element containing all of the active teams at v_k. If so, the recursion has locally reached a base case and planning cannot be further delegated to smaller subplanners. Therefore, the subplanner calls GETNEIGHBORS to generate the limited neighbors by enumerating all possible actions. Otherwise the rCMS subplanner iterates over each conflict set element C^i_k and calls the appropriate subplanner to find a solution to the subproblem defined by C^i_k. Once a solution has been computed for each subproblem, a single limited neighbor is generated by combining the next step specified by each partial solution with the next step along the individual policy for each team not part of a subproblem. If any subproblem does not have a solution from v_k, then the goal is not reachable from v_k and GETNEIGHBORSRECURSIVE returns an empty set.

CMS and rCMS are complete and will return minimal cost paths. The proofs generally follow those for M* (Wagner and Choset 2011), and are given in full in (Wagner 2015).

Results

We validate the performance of CMS and rCMS in simulation. Teams of three robots each must move large, rigid, rectangular loads from fixed initial configurations to fixed destination. The load may not contact any object aside from the robots carrying the load. The constraint manifolds corresponding to the tasks are diffeomorphic to SE(2). When a robot is in a singleton team it moves on an 8-connected grid

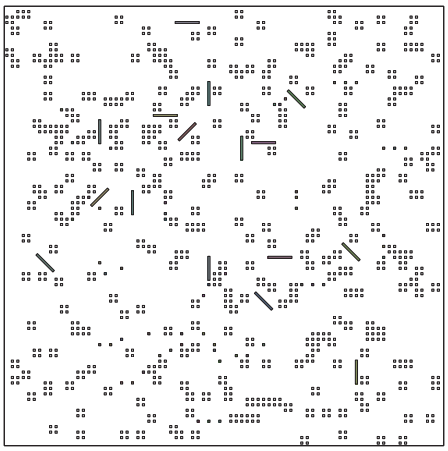

Figure 2: Typical random environment with 20% obstacle coverage. The empty squares are obstacles. The colored squares are individual robots, and the long rectangles teams of three robots carrying a heavy load. The configuration of teams is taken from halfway through a path found by inflated $rCMS$ with $\epsilon = 3$ for a problem involving 102 robots in 34 teams.

where waiting as well as vertical and horizontal movement costs 1, while diagonal movements cost $\sqrt{2}$. A robot may wait for zero cost at its final destination or at the start configuration of its next task if the other robots assigned to the task are not yet in position. The manifold graphs for teams of three robots carrying a load restrict the central robot to a similar 8-connected grids. The robots at either end of the load can rotate about the central robot in increments of $\pm 45°$. Rotation incurs cost 3, while translation incurs 3 times the cost of a single robot translating. To simplify the problem, the loads are assumed to be removed from the workspace after being delivered by a team.

Tests were run in randomly generated 80x80 grid worlds. Approximately 20% obstacle coverage was generated by randomly placing 320 2x2 obstacles in the workspace with overlaps permitted. The robot and load initial and final goal configurations were randomly generated, subject to the constraint that each task be feasible in isolation (Figure 2). Robots were grouped in sets of three, with every such team assigned 1, 3, or 5 tasks[3]. Simulations included up to 120 robots. 50 random trials were generated for each set of test parameters. CMS and rCMS were both run with a heuristic inflation factor of 3. Each trial was given 5 minutes to find a solution before the trial was marked as a failure (Figure 3). All simulations were implemented in Python and run on a Intel Core i7 processor clocked at 3.30 GHz.

rCMS dramatically outperformed CMS with the success rate of rCMS on problems involving 3 to 5 tasks per team equivalent to the performance of CMS with only a single task per team (Figure 3). The environments were relatively

[3]We tested randomly assigning the robots to each task, which increased the time the robots spent waiting for other robots, and made the problem easier.

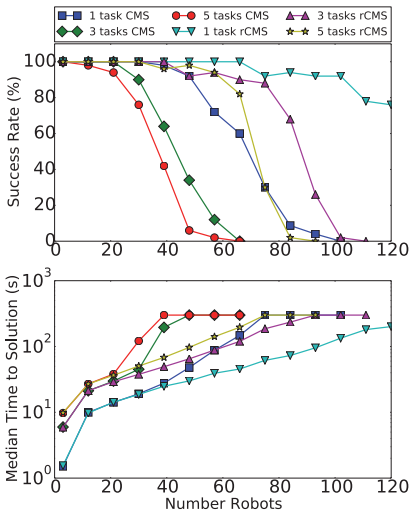

Figure 3: Comparison of CMS and rCMS performance. All trials used an inflation factor of 3. The top plot shows the percentage of trials for which a solution was found within 5 minutes, while the bottom plot gives the median time until a solution was found.

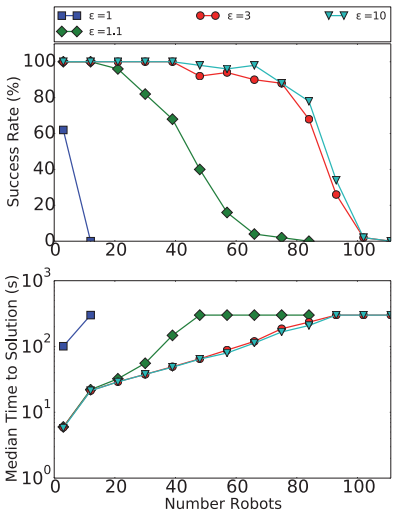

Figure 4: Performance of rCMS with varying inflation factors. Each team was assigned 3 tasks, in worlds with 20% obstacle coverage. The top plot shows the percentage of trials for which a solution was found within 5 minutes, while the bottom plot gives the median time until a solution was found.

large, which meant there were often multiple independent sets of interacting teams. rCMS was therefore able to find independent subproblems, which could be solved more efficiently.

We then tested the impact of varying the inflation factor of rCMS (Figure 4), with each team assigned 3 tasks. As expected, rCMS was unable to solve anything but the simplest problems optimally ($\epsilon = 1$). When $\epsilon = 1$ most collisions will force M* to re-expand the root vertex of the search tree, where the coupled set has maximal size. Setting $\epsilon = 1.1$ sets a very soft limit to backtracking; to prevent incurring one extra unit of cost, rCMS would be willing to backtrack approximately $10/n$ steps in the search tree, where n is the number of teams that have not reached their final goal. Limiting backtracking effectively decouples collisions that take place at different times, significantly reducing the cost of finding a solution. When the inflation factor is increased to $\epsilon = 3$, rCMS will come close to greedily minimizing the heuristic value, primarily backtracking to resolve dead-ends rather than reduce path cost. As a result, there is a substantial improvement in success rate, and rCMS is able to solve some problems involving 102 robots in 34 teams, where as with $\epsilon = 1.1$ rCMS was only able to solve systems with 75 robots in 25 teams. Increasing ϵ to 10 results in little change in performance, as rCMS was already operating in a close to greedy manner.

Conclusion and Future Work

We presented a new formulation for CMS that uses a task graph to reason about the dependencies between teams. The reformulated algorithm is simpler than the existing formulation, and allows more general cost functions and task schedules. We then presented rCMS, a variant of CMS that decouples planning for disjoint sets of interacting teams. Simulation studies showed that rCMS significantly outperforms CMS.

There are a number of avenues for future work. CMS has been implemented for only a single cooperative task: multiple mobile robots carrying a single object. In the future, we will pursue techniques for automatically constructing the constraint manifold (Berenson et al. 2009) (Siméon et al. 2004) (Cohen, Chitta, and Likhachev 2013) (Sucan and Kavraki 2011) to apply CMS to more types of tasks. Of special interest is the task of cooperatively carrying an object through an environment that is sufficiently cluttered that robots must disconnect and reconnect to the object to clear obstacles (Lindzey et al. 2014), potentially resting the load on the ground temporarily.

To simplify the problem, we assumed that actions of teams do not impact the environment, even when the teams move large objects. We believe that this assumption can be removed by treating environmental features that may be manipulated by robots as dummy teams that contain no robots. CMS will then naturally account for dependencies between the teams that manipulate the environmental features and the teams that may be impeded by the environmental features.

Finally, CMS currently assumes that each team can wait at its goal configuration for zero cost, which is necessary to properly decouple planning between teams. We believe identifying lower bounds on arrival times of different teams at their goal configurations will allow generalization to cost functions that penalize robots for waiting at their goals.

This work was supported by ONR Subcontract to the Applied Physics Lab entitled ”Autonomous Unmanned Vehicles Applied Research Program” Prime Contract Number N00024-13-D-6400

References

Alonso-Mora, J.; Knepper, R. A.; Siegwart, R.; and Rus, D. 2015. Local Motion Planning for Collaborative Multi-Robot Manipulation of Deformable Objects. In *IEEE International Conference on Robotics and Automation*, 5495–5502.

Ayanian, N., and Kumar, V. 2010. Decentralized feedback controllers for multi-agent teams in environments with obstacles. *IEEE Transactions on Robotics* 26(5):878–887.

Ayanian, N. 2011. *Coordination of Multirobot Teams and Groups in Constrained Environments : Models , Abstractions , and Control Policies*. Ph.D. Dissertation, University of Pennsylvania.

Belta, C., and Kumar, V. 2001. Motion generation for formations of robots: a geometric approach. In *IEEE International Conference on Robotics and Automation*, number 3, 1245–1250.

Berenson, D.; Srinivasa, S. S.; Ferguson, D.; and Kuffner, J. J. 2009. Manipulation planning on constraint manifolds. In *IEEE International Conference on Robotics and Automation*, volume i, 625–632. Ieee.

Bhattacharya, S.; Likhachev, M.; and Kumar, V. 2010. Multi-agent path planning with multiple tasks and distance constraints. In *IEEE International Conference on Robotics and Automation*, 953–959. Ieee.

Burridge, R.; Rizzi, A. A.; and Koditschek, D. E. 1999. Sequential composition of dynamically dexterous robot behaviors. *International Journal of Robotics Research* 18(6):534–555.

Cohen, B.; Chitta, S.; and Likhachev, M. 2013. Single- and dual-arm motion planning with heuristic search. *The International Journal of Robotics Research*.

Desai, J. P., and Kumar, V. 1999. Motion planning for cooperating mobile manipulators. *Journal of Robotic Systems* 16(10):557–579.

Desaraju, V. R., and How, J. P. 2012. Decentralized path planning for multi-agent teams with complex constraints. *Autonomous Robots* 32(4):385–403.

Felner, A.; Goldenberg, M.; Sharon, G.; Stern, R.; Beja, T.; and Holte, R. C. 2012. Partial-Expansion A * with Selective Node Generation. In *AAAI Conf*, 471–477.

Graham, R. L.; Lawler, E. L.; Lenstra, J. K.; and Kan, a. H. G. R. 1979. Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey. *Annals of Discrete Mathematics* 5(C):287–326.

Hu, T. C. 1961. Parallel Sequencing and Assembly Line Problems. *Operations Research* 9(6):841–848.

Koga, Y., and Latombe, J.-C. 1994. On multi-arm manipulation planning. In *IEEE International Conference on Robotics and Automation*, 945–952. San Diego, CA: IEEE Comput. Soc. Press.

Leonard, N. E., and Fiorelli, E. 2001. Virtual Leaders , Artificial Potentials and Coordinated Control of Groups. In *IEEE Conference on Decision and Control*, number December, 2968–2973.

Lewis, M. A., and Tan, K.-h. 1997. High Precision Formation Control of Mobile Robots Using Virtual Structures. *Autonomous Robots* 403:387–403.

Lindzey, L.; Knepper, R. A.; Choset, H.; and Srinivasa, S. S. 2014. The Feasible Transition Graph: Encoding Topology and Manipulation Constraints for Multirobot Push-Planning. In *Workshop on the Algorithmic Foundations of Robotics*, 16.

Pohl, I. 1973. The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In *International Joint Conference on Artificial intelligence*, 12–17. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Rus, D.; Donald, B.; and Jennings, J. 1995. Moving Furniture with Teams of Autonomous Robots. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 235–242.

Sharon, G.; Stern, R.; Felner, A.; and Sturtevant, N. R. 2012. Meta-Agent Conflict-Based Search For Optimal Multi-Agent Path Finding. *Symposium on Combinatorial Search* 97–104.

Siméon, T.; Cortés, J.; Sahbani, A.; and Laumond, J.-P. 2004. A general manipulation task planner. In Boissonnat, J.-D.; Burdick, J.; Goldberg, K.; and Hutchinson, S. A., eds., *Workshop on the Algorithmic Foundations of Robotics*, volume 7 of *Springer Tracts in Advanced Robotics*, 311–327. Springer Berlin Heidelberg.

Spears, W., and Gordon, D. 1999. Using artificial physics to control agents. In *International Conference on Information Intelligence and Systems*, 281–288.

Standley, T. 2010. Finding Optimal Solutions to Cooperative Pathfinding Problems. In *AAAI Conference on Artificial Intelligence*.

Sucan, I. A., and Kavraki, L. E. 2011. On the advantages of task motion multigraphs for efficient mobile manipulation. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 4621–4626. IEEE.

Wagner, G., and Choset, H. 2011. M*: A Complete Multirobot Path Planning Algorithm with Performance Bounds. In *IEEE International Conference on Intelligent Robots and Systems*.

Wagner, G.; Kim, J. I.; Urban, K.; and Choset, H. 2015. Constraint Manifold Subsearch for Multirobot Path Planning with Cooperative Tasks. In *IEEE International Conference on Robotics and Automation*, 6094–6100.

Wagner, G. 2015. *Subdimensional Expansion: A Framework for Computationally Tractable Multirobot Path Planning*. Phd, Carnegie Mellon University.

Yang, P.; Freeman, R. A.; and Lynch, K. M. 2008. Multi-agent coordination by decentralized estimation and control. *IEEE Transactions on Automatic Control* 53(11):2480–2496.