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 $\epsilon$-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) (Feller 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
the configuration graph of \( \tau^j \) at which task execution can start, \( v^j_s \) is the goal vertex in the configuration graph of \( \tau^j \) at which the task can be completed, and \( \mathcal{C}^j \) is a set of task constraints that the robots in \( \tau^j \) 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 incur 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 \( \mathcal{M}^t \) of team \( \tau^t \) with the weighted, directed, manifold graph \( G^t = \{ V^t, E^t \} \). Vertices \( v^t \in V^t \) represent valid team configurations, while edges in \( E^t \) 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.
The teams that precede

If team τₐ is assigned to team τᵢ, τᵢ is involved in a conflict with another team then τᵢ can take a different path from vᵢₖ to vᵢₗ.

The teams that precede τᵢ can alter their paths so that τᵢ forms earlier or later, altering the position of τᵢ relative to other moving teams.

3. The other input teams for the transition action that completes τᵢ can take shorter, but potentially more expensive, paths to reach their goal vertices earlier, allowing τᵢ to take the transition action before the conflict would occur.

Note that case 3 is only relevant if τᵢ 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 τᵢ. 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 τᵢ using a modified task graph. First note that case 3 means that the team dependencies change once τᵢ reaches its goal vertex for the first time. Therefore, τᵢ is labeled τᵢₗ before it reaches its goal vertex, and τᵢ₀ₗ after it has reached its goal vertex. To simplify later discussion, τᵢ is used to denote both τᵢₗ and τᵢ₀ₗ in cases where the distinction is not relevant. CMS then defines the task graph as a directed tripartite graph Gₜₖₜ = {Vₚₜ, Vₛₜ, Eₜₖₜ}. The vertices τᵢₖiare ∈ Vₚₜ and τᵢ₀ₗ ∈ V₄ₜ are teams before and after reaching their goal vertices, respectively. The transition vertices τᵢₗ ∈ Vₚₜ represent transition actions. In the special case where vᵢₖ = vᵢₗ, for instance because τᵢ is a single robot that must wait for another team to arrive to form its next team, the task graph contains τᵢₗ but not τᵢₜₗ. Let vₜₕ ∈ Vₚₜ ∪ V₄ₕ ∪ V₄ₚ 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 vₜₕ is dominated by vₜₕ ∈ Gₜₖₜ, denoted vₜₕ ≤ vₜₕ, if there is a path in Gₜₖₜ from vₜₕ to vₜₕ or vₜₕ = vₜₕ. Similarly, vₜₕ ≤ vₜₕ if vₜₕ ≤ vₜₕ ∧ vₜₕ ≠ vₜₕ. Two vertices are incomparable if there is no path between them in Gₜₖₜ. 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 τⱼ that must finish their tasks before τᵢ can start, which corresponds to τⱼ < τᵢ. If case 3 is relevant, then τᵢ = τᵢ₀ₗ ∈ V₄ₜ, and its successor is a transition vertex which dominates all of its input teams. CMS can thus use the successor of τᵢ to determine the relevant teams, as if τᵢ ∈ Vₚₜ its only successor is τᵢ₀ₗ whose only direct predecessor is τᵢ. We therefore conclude that if τᵢ is involved in a collision, then CMS must consider alternate paths for all teams τⱼ such that τⱼ < successor (τᵢ) or τⱼ = τᵢ, where successor (τᵢ) is the successor of τᵢ in the task vertex. τᵢ = τᵢ₀ₗ condition is necessary for terminal teams that have no successors in the task graph, such as τᵢ₀ₗ in Figure 1.

---

2. τᵢ may temporarily move away from vᵢₗ 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

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 G_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 (τ^i, v^i), 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 T^act denote the set of teams performing their task at v_k. Edges in G correspond to transitions between teams.

CMS begins by computing a separate individual policy φ^i : V^i → 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^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.

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 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 ∩ G_k ≠ ∅ then
8:       continue
9:     v_k.back_prop_set.add(v_l)
10: BACKPROPAGATE(v_k, C_k, open_list)
11: if v_k.g + EDGE_COST(v_k, v_l) ≥ v_l.g ∨ v_l.closed then
12:     continue
13:     v_l.g ← v_k.g + EDGE_COST(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 COUPLED_SET(C) ⊂ COUPLED_SET(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: if v_l ∈ v_k.back_prop_set do
26:   BACKPROPAGATE(v_l, C_k, open_list)

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 τ^i < successor (C) ∨ τ^i ∈ C, where successor (C) is the set of successors of teams in C.

CMS functions by alternating between running A* search on a low-dimensional search graph and expanding the search graph to provide 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 G_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 (τ^i, v^i), 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 T^act denote the set of teams performing their task at v_k. Edges in G correspond to transitions between teams.

CMS begins by computing a separate individual policy φ^i : V^i → 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^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.

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 ← $\emptyset$  
4: // Generate possible actions one team at a time  
5: for $(\tau^i, v^i_k) \in v_k$ do  
6: // Temporary variable  
7: add_step ← $\emptyset$  
8: // Iterate combinations of actions for earlier teams  
9: for neib ∈ neighbors do  
10: if $\exists \tau^j \in$ neib, $\tau^i \leq \tau^j$ then  
11: // $\tau^j$ already assigned a transition action  
12: add_step.append(neib)  
13: continue  
14: add_step.append(neib ∪ GETPOLICY($v_k$, $\tau^i$, neib))  
15: // Consider all actions for teams in the coupled set  
16: if $\tau^i \in \Gamma_k$ then  
17: for $v^j_k \in$ OUTNEIGHBORS($v_k$) do  
18: add_step.append(neib ∪ $\lbrace (\tau^i, v^j_k) \rbrace$)  
19: neighbors ← add_step  
20: return neighbors  

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

If the edge connecting $v_k$ to $v_f$ is collision free CMS marks $v_k$ as a predecessor of $v_f$ by adding $v_k$ to the back-propagation set (back_prop_set) of $v_f$. Then BACKPROPAGATE is called to add the conflict set of $v_f$ to $C_k$. CMS then updates the g-value of $v_f$ and adds $v_f$ 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 $\tau^i$ 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 $\Gamma_k$ containing all the teams that could alter the path taken by the teams in the conflict set (Lemma 1)

$$\Gamma_k = \lbrace \tau^i \in T_k^{act} | \tau^i < \text{successor} (C_k) \lor \tau^i \in C_k \rbrace,$$  \hspace{1cm} (1)

where successor ($C_k$) 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).

Algorithm 4 GetPolicy

1: // Generate action taken by $\tau^i$ when following its individual policy at $v_k$, including taking transition actions  
2: function GETPOLICY($v_k$, $\tau^i$, $\tau^j$, neib)  
3: ret ← $\emptyset$  
4: // Take the transition action if possible, else follow individual policy  
5: if CANTRANSITION($\tau^i$, $v_k$, neib) then  
6: for $\tau^i \in$ OUTPUTTEAMS($\tau^j$) do  
7: ret ← ret ∪ $\lbrace (\tau^i, v^j_k) \rbrace$  
8: else  
9: ret ← $\lbrace (\tau^i, \phi(v^j_k)) \rbrace$  
return ret  

10: function CANTRANSITION($\tau^i$, $v_k$, neib)  

Returns True if every input team for the next transition action of $\tau^i$ 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($\tau^i$)  

12: Returns the output teams of the transition vertex for which $\tau^i$ is an input team.

As an example of how coupled sets are computed consider the case of the task graph given in Figure 1. If teams $\tau^1$ and $\tau^2$ were to collide before $\tau^1$ and $\tau^2$ reached their goal configuration, then the conflict set would be $\lbrace \tau^1_{tm}, \tau^2_{tm} \rbrace$. CMS may then backtrack in the search tree to consider alternate paths from a vertex $v_f$ with the active teams $\lbrace \tau^1_{tm}, \tau^2_{tm}, \tau^3_{tm} \rbrace$. The coupled set of $v_f$ would be the teams dominated by $\lbrace \tau^1_{gl}, \tau^2_{gl} \rbrace$, which are $\lbrace \tau^1_{tm}, \tau^2_{tm} \rbrace$. However, if CMS had found a potential collision between $\tau^1$ and $\tau^2$ after $\tau^2$ had reached its goal configuration then the conflict set would be $\lbrace \tau^1_{tm}, \tau^2_{gl} \rbrace$. If CMS then backtracked to the same vertex $v_f$, the coupled set would be the teams dominated by $\lbrace \tau^1_{gl}, \tau^2_{gl} \rbrace$, which are $\lbrace \tau^1_{tm}, \tau^2_{tm} \rbrace$. $\tau^3$ would be added to the coupled set because $\tau^3$ might still avoid the collision with $\tau^1$ by taking the team formation action earlier. $\tau^2$ can only take the transition action when $\tau^3$ is also at its goal configuration, so $\tau^3$ 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 $\epsilon$ 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 $\epsilon$-suboptimal, which means that it may return a path that costs up to $\epsilon$ 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
1: function GETNEIGHBORSRECURSIVE(vk)
2: if NUMELEMENTS(Γk) = 1 ∧ Γk = Tk then
3: return GETNEIGHBORS(vk)
4: // Initialize a vertex with an empty coordinate
5: vtl = ∅
6: // Compose partial solutions for each subproblem
7: for Ck ∈ Ck do
8: query_vertex = \{ (τi, vl) | τi ∈ Γk \}
9: vtl ← vtl ∪ GETSTEP(Ck, query_vertex)
10: if (τi, vl) ∈ vtl then
11: continue
12: vtl.append(GETPOLICY(vk, τi))

14: 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 (τi, vl) pair is generated, test if τi ∈ Vtm and vl = vtl. If so, replace τi with τgi. If τgi is not in the current subproblem, note the action but do not add (τgi, vl) 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 τi was found to conflict with τj, then CMS would consider all possible actions for the set of teams {τi, τj}. If each team had 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 τi and τj independently of the collision between τi and τj, 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 combination 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 contains 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 Tres (C).

\[ T_{res} (C) = \{ v_{task} | v_{task} \prec successor (C) \} \]

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 τi is in the subproblem task graph but τi is not, then the transition action (respectively the action taking τi to its goal vertex) remains part of the solution, but the output teams (resp. τgi) 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 criterion 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 τi and τgi, and a second conflict between τj and τj, leading to the conflict set \{ (τi, τgi), (τj, τj) \}. 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 \( \tau_{tm}^\ell \) 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 \( \{ r_{tm}^1, r_{tm}^5, r_{tm}^6 \} \) the two subproblems are effectively disjoint. However, if rCMS backtracks to a vertex \( v_k \) whose active teams are \( \{ r_{tm}^1, r_{tm}^5, \tau_{tm}^7 \} \) \( \tau^4 \) would be in the subproblems associated with both conflict set elements, thus the conflict set elements must be merged at \( v_k \).

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 separate 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 \( C_k^\ell \) and calls the appropriate subplanner to find a solution to the subproblem defined by \( C_k^\ell \). 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 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 \( \pm 45^\circ \). 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 moved 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. 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 an 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

---

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image.png}
\caption{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.}
\end{figure}

---

\[\text{We tested randomly assigning the robots to each task, which increased the time the robots spent waiting for other robots, and made the problem easier.}\]
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 $\epsilon$ 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 couples 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


San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.


