Planning in Constraint Space: Automated Design of Functional Structures

Can Erdogan Mike Stilman

***Abstract*—On the path to full autonomy, robotic agents have to learn how to manipulate their environments for their benefit. In particular, the ability to design structures that are functional in overcoming challenges is imperative. The problem of automated design of functional structures (ADFS) addresses the question of whether the objects in the environment can be placed in a useful configuration. In this work, we first make the observation that the ADFS problem represents a class of problems in high dimensional, continuous spaces that can be broken down into simpler subproblems with semantically meaningful actions. Next, we propose a framework where discrete actions that induce constraints can partition the solution space effectively. Subsequently, we solve the original class of problems by searching over the available actions, where the evaluation criteria for the search is the feasibility test of the accumulated constraints. We prove that with a sound feasibility test, our algorithm is complete. Additionally, we argue that a convexity requirement on the constraints leads to significant efficiency gains. Finally, we present successful results to the ADFS problem.**

I. Introduction

With increasing structural agility and computation power, robotic agents have become viable options in real-life problems which cover a wide range of applications such as search and rescue missions, space exploration and military operations. Despite this surge of robotics, most robotic applications today have extensive human intervention in the control of robot behavior. Only a few agents function fully autonomously and always in well controlled environments.

In this paper, we focus on the problem of automated design of functional structures (ADFS). The ability to use the objects in the environment to one's benefit and construct meaningful, functional structures is an imperative step towards full autonomy. However, to be dependable, such planners have to be complete, in the sense that they have to find a solution to a given challenge if one exists. If such a guarantee can be provided, a myriad of applications can benefit from these planners such as building bridges autonomously in war zones, creating and realizing designs to close deep ocean valves, as in natural disasters, and the autonomous construction of human environments in space exploration.

In an automated design process, a planner needs to make two types of choices: (1) discrete choices to determine the involvement of the objects in the environment, and (2) continuous domain choices to determine the locations of the objects in the design. The discrete choices have to be made on which objects to include in the design and what role they would play, an abstract, rough description of their relationship to other objects (i.e. on the ground or on another object). On the other hand, a continuous choice is on where *exactly* an object o_i is and this variable can take an infinite number of values: $x_i = 0.12$ or $x_i = 0.1294$. Usually, continuous choices are discretized, i.e. x_i can be either 0.12 or 0.13, but not anything in between. However, such a limitation discounts all the successful designs that have $x_i = 0.125$ and invalidates the completeness of a planner.

The authors are with the Center for Robotics and Intelligent Machines at the Georgia Institute of Technology, Atlanta, GA 30332, USA. Email: cerdogan3@gatech.edu, mstilman@cc.gatech.edu

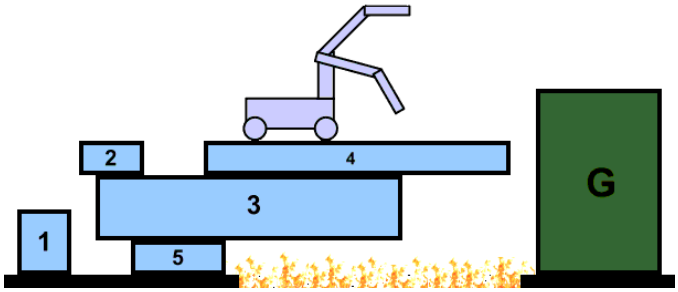


Fig. 1. An example of a robotic agent having designed and built a stable structure to climb the obstacle G and avoid the hazardous flaming region. Objects o_1, o_3 and o_5 have 2 units of mass, where objects o_2 and o_4 have 6 and 3 units. The robot weighs 0.5 unit.

In the domain of automated design, the planner has to consider both the discrete and continuous choices to design a structure that is both stable and can be traversed to reach the problem goal. An example challenge may include crossing over hazardous areas and climbing up heights where structures such as bridges and stairs need to be designed using the available objects. Figure 1 demonstrates such an example output of our work. The structure is guaranteed to be stable as the agent traverses the structure to climb the goal structure G. Note that the consideration of object masses enables a design where a small but heavy object, o_2, is used to counterbalance the weight of the other objects.

A Complete Planner for Continuous Domains

The importance of level of discretization in efficiently limiting the space of solutions has been studied carefully. A well-known approach is to create a planning graph for the discretized space. In the graph, a node is a candidate solution and an edge between two nodes n_1, n_2 represents an incremental change to node n_1 to transform it to its neighbor n_2. The problem then is reduced to finding a path from the start node n_0, i.e. an empty design, to a goal node n_G which satisfies the challenges of the domain. Such a path would dictate which

actions should be taken to reach the desired goal. Although the reduction to a search problem is effective, finer discretizations lead to more number of neighbors for each node. Thus, in addition to being incomplete, the discretization approach also suffers from having a large search space.

Assume that the planner already knows which objects to place. Then, instead of limiting the space of object configurations with discretization, the problem can be reformulated as a *feasibility problem* where the challenges of the domain are expressed as constraint equations whose variables denote the locations of the chosen objects. In general, once the correct actions are taken and the constraints are introduced to the system, a sound feasibility test would return the desired solution configuration if one exists. To this end, we frame the design problem as a discrete search where the choices are only on the roles of the objects in the design and use feasibility tests to evaluate goal nodes. In this framework, a new action, i.e. a discrete choice, adds new constraints to the space of feasible configurations. For instance, if $o1$ is placed on $o2$, we have the constraint '$y1 - y2 = 0.5(h1 + h2)$' which expresses the fact that the height difference between the centers of the two objects has to be half of their total height. The two problems with the discretization approaches are avoided in this manner.

With our proposed approach, the high dimensional continuous problem is reduced to finding a succession of actions that would induce the correct constraints that represent a functional design. The reformulation has two benefits. First, given a sound feasibility test and a complete planner in discrete space, the planner is guaranteed to find a solution if one exists. Second, if constraints have simple forms, such as linear equality and inequalities, then fast feasibility tests can be used for performance. Thus, *planning in the constraint space* can be guaranteed to be complete and can lead to significant progress in efficiency over randomized or discretized approaches.

In the rest of this paper, we will formulate our approach and present results in the domain of automated design. We make the following contributions:

1) The reformulation of continuous space problems in terms of discretized actions in constraint space
2) A complete planner for constraint space problems
3) The problem definition for the domain of autonomous design of functional structures
4) The application of the approach to the ADFS domain

II. Related Work

The ability to realize precise actions is important in several areas of robotics and planning, such as medical robotics and autonomous design. To consider more precise actions, classical planning approaches discretize at higher levels of detail. However, the increase in the higher ordered number of such detailed actions leads to efficiency problems.

The essential shortcoming of classical planning approaches in continuous domains is due to the idea of commitment. In classical progression or regression search, the planner has to commit to taking an action and evaluate all the possible outcomes. In hierarchical task networks, with increasing problem details, the planner has to commit to more number of problem decompositions. Although the SAT planner and the graphPlan algorithms avoid the problem of commitment, they still have to consider increased number of combinations of actions [8].

The common approach to accommodate non-convex high dimension continuous domains in the motion planning area is the rapidly-exploring random tree (RRT) algorithm [6]. The baseline algorithm creates a tree in the problem space, with the initial configuration as the root node, by choosing random configurations and incrementally extending edges towards them from the closest neighbor node on the tree. Despite its clear advantages in efficiently covering the problem space, the RRT suffers from the same source of shortcoming as classical planning algorithms. The commitments to the random configurations and the closest neighbors can lead to (1) unrealistic and high costly paths [1] or (2) long run times to find precise solutions (i.e. paths in cluttered worlds) [4].

A. Optimization in Planning

In this work, we further the ability of classical planning approaches to accommodate high dimensional continuous domains by introducing constraint optimization as a method to evaluate candidate states. Particularly, we search for object configurations in a convex continuous domain, a relaxation of the non-convex problem specification of RRT's. This relaxation allows the use of efficient optimization techniques.

The key idea is that with each discrete abstract action choice, we partition the convex space and evaluate whether there is a feasible solution in the partitioned spaces. After eliminating those without feasible solutions, we continue making action choices until we reach a state that satisfies the abstract goal specifications and obtain a feasible continuous solution to the problem. Note that the convexity assumption allows for global feasibility searches in the entire problem manifold and so, guarantees the completeness of the planner.

Global and local optimization techniques have been heavily used in the motion planning and manipulation areas. Khatib showed that the design of a manipulator can be optimized for by minimizing a cost function with respect to kinematic, dynamic and actuator design parameters [5]. Buss et al. reformulated the problem of correct grasping force for a robotic hand in a constrained convex optimization setup and developed real-time gradient flow algorithms [2]. To enable the manipulation of constrained objects such as doors and shelves, Stilman proposed a local gradient search method in the joint space of the manipulator where the gradient is derived from the task constraint errors [9]. Note that most of the work in motion planning is strictly in non-convex configuration spaces and do not necessarily partition into subproblems.

The optimization of continuous variables along with discrete constraints has also been studied in operations research and scheduling literature. Vidal and Geffner [10] propose a method that uses temporal constraints to make powerful inferences about the dependencies of the actions. In fact, our work resembles the branch-and-bound constraint satisfaction approach adopted for scheduling problems [3].

III. Problem Definition

Let O be the set of objects in the environment. Each object $o_i \in O$ has three properties, $\{w_i, h_i, m_i\}$, which stand for the width, height and mass respectively. The configuration $\vec{x}_i$ of an object $o_i \in O$ is its position $\{x_i, y_i\}$. The orientations are not included as continuous parameters to demonstrate the efficiency of linear optimization tools. Otherwise, the rotation angles with nonlinear trigonometric functions would induce nonlinear constraints and invalidate linear approaches.

A. State and action representations

The idea is to represent a design with a set of abstract propositions. For example, $\{On(o_0, o_1), On(o_1, o_2)\}$ represents a design with object o_0 on o_1 and object o_1 on o_2. Note that this representation is discrete and does not account for the specific locations of the objects. In the initial state of the problem, the robot is always on the ground, represented with the literal $At(ground)$. The planner has to find a path of actions which ends up at a state where the literal $At(goal)$ exists. The definition of the $goal$ location depends on a specific problem. For example, it can be the top of a climbed obstacle or the area beyond an hazardous region.

The set $A(O)$ is the set of all possible actions in the environment such as placing an object on the ground or moving from one object to another. An action $A_i \in A(O)$ is defined with three sets of literals: preconditions $\mathbf{P}(A_i)$, the add effects $\mathbf{A}(A_i)$ and the delete effects $\mathbf{D}(A_i)$. For an action to be taken in a state S, all the literals in $\mathbf{P}(A_i)$ must exist in S. After an action is taken, the literals in $\mathbf{A}(A_i)$ are added to the state and those in $\mathbf{D}(A_i)$ are removed.

In Table I, we present the set of actions in the ADFS domain. Note that when subsequent actions that move the robot are taken, we assume that the robot traverses the distance between the two locations. For instance after climbing above from o_1 to o_2 and before jumping from o_2 to o_3, we assume that the robot traverses the length of o_2 successfully.

WalkFromGround(o_1)	Climbs to o_1 from ground given OnGround(o_1)
WalkAbove(o_1,o_2)	Climbs to o_2 from o_1 given On(o_2,o_1)
WalkBelow(o_1,o_2)	Goes down to o_2 from o_1 given On(o_1,o_2)
Jump(o_1,o_2)	Drives over a gap from o_1 to o_2
PutOn(o_1,o_2)	Puts o_1 on o_2 given Used(o_2) and Unused(o_1)
PutGround(o_1)	Puts o_1 on ground given Unused(o_1)

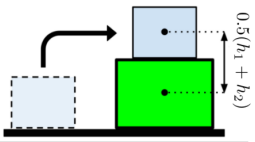
TABLE I
The descriptions of the ADFS actions

The specification of constraints that govern the continuous configuration variables are in the add effects of an action. Figure 2 demonstrates two examples of actions along with the constraints that they induce. Using the discrete, abstract representation of designs, the planner performs a search, looking for a goal state where all the state, the problem and stability constraints are satisfied.

B. Space of solutions

A solution is a composition of two lists $\{X_{O^*}, A_{O^*}\}$. The list X_{O^*} is a list of configurations $\{\vec{x}_i | o_i \in O^*\}$ where $O^* \subset O$ is the set of objects that were used in the design. It represents the final goal positions of the objects after optimization. Let $\mathbb{P}(A(O))$ be the set of permutations of feasible actions using objects O. Then, the list $A_{O^*} \in \mathbb{P}(A(O^*))$ is a list of actions that leads to the building of the final design X_{O^*} and the movement of the robot from its start state with the literal $At(ground)$ to the goal state with the literal $At(goal)$.

Name: PutOn(o_1,o_2)
Pres: {Unused(o_1), Used(o_2), ¬At(o_2,_)}
Adds: {Used(o_1), On(o_1,o_2), Increment($nObjs[o_1]$),
Set($onID[o_2][o_1]$,$nObjs[o_1]$),
Eq($y_1 - y_2 = 0.5(h_1 + h_2)$)}
Dels: {Unused(o_1)}



Name: Jump(o_1,o_2)
Pres: {At(o_1), Used(o_1), Used(o_2), CanGo(o_2)}
Adds: {At(o_2), Complex($leftSpace$),
InEq($x_2 - x_1 \leq 0.5(w_1 + w_2) + max_x$),
InEq($x_2 - x_1 \leq -0.5(w_1 + w_2)$),
InEq($y_2 - y_1 \leq 0.5(h_1 - h_2) + max_y$),
InEq($y_2 - y_1 \leq -0.5(h_1 + h_2)$)}
Dels: {Unused(o_1), CanGo(o_2)}

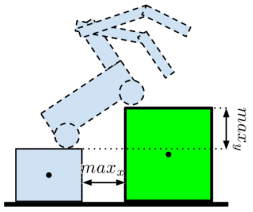


Fig. 2. The example definition for two actions, PutOn(o_1,o_2) and Jump(o_1,o_2), which put one object on another and drive the robot over a gap between two objects, along with their effects in the environment.

IV. Approach

In the following, we discuss our approach in detail, particularly: (1) the merits of having actions with literal state effects and constraints, (2) the search strategy in constraint space, (3) a simple example application, and (4) the pseudo-code.

A. Partitions of the Configuration Space

The reduction of the high dimensional continuous problem to a discrete search problem is possible due to the two types of effects that actions have. First, each action has logical state literals that are built to describe the physical properties of the structure and the location of the robot. These statements specify a goal state for the search problem and in our case, a state with the literal $At(goal)$ is a candidate goal state.

It should be noted that this description of a goal state and the subdivision of the problem into boolean literals do not have to generalize to every high dimensional continuous problem and this is why we focus only on those where the constraints that define the goal subspace can be semantically expressed as atomic actions. For instance, the well studied metric simultaneous localization and mapping problem that has millions of variables to solve for can not be expressed as a search problem with discrete actions.

In addition to the logical state literals, each action induces additional constraints over the space of goal configurations. This framework has two fundamental effects. First, given that a goal state has the desired goal literals and has a feasible configuration for the constraints it contains, we are guaranteed to find a solution to the search problem if a feasible solution to the original continuous problem exists and the feasibility test is sound. This is completeness.

Secondly, each time an action is added, the planner checks for the feasible goal subspace. If the last action introduces

constraints that are in violation of the domain constraints, e.g., requires a jump of 10 meters, then the outcome state and all its future children can be discarded. In other words, each action partitions the space of configurations. A subspace without feasible results is discarded. The recursive partitioning of the space continues until the goal literals are satisfied at a state and the final space defined by the state constraints is not empty. Any value from that final space is a goal configuration and the feasibility test would return one if it exists.

B. Search in Constraint Space

Having established that the desired goal configuration is searched for in the constraint space by adding actions which induce new constraints to the system, we discuss the search methodology. In this work, we use the backtracking forward planner with uninformed heuristics to order future actions.

Remember that the goal is to search for a node that satisfies the goal literals and its constraints, starting from some initial node n_0. The backtracking forward planner is essentially a depth first search algorithm where at a node n_i with the neighbors (children) $\{n_{i1}, ...n_{im}\}$, the search commits to going to each child n_{ij} (e.g., depth first) and backtracks if none of the paths following n_{ij} lead to some goal state n_g. The term 'forward' planner defines that a child n_j of a node n_i can be generated from n_i by applying some action a_k (i.e. n_j = Apply(n_i, a_k). To choose which child nodes to inspect first, we use a predefined action ordering, an uninformed heuristic since it does not use any node information.

A note on backward planners and informed heuristic methods: The 'backward' planner defines a node n_j to be child of n_i if an action a_k can be taken to generate n_i (i.e. n_i = Apply(n_k, a_k)) - an exact opposite of the 'forward' planner. When action definitions only have effects on state literals, this method is preferred if the goal state has less neighbors. However, in our framework, it would invalidate the partitioning approach. Imagine the counterbalancing in Figure 1. It can not be generated because in the predecessor state, before the counterbalance, the state would be unbalanced and discarded. The child would never be generated. We have experimented with extensions of backward planners to accommodate searches in the constraint space but have decided to present our approach with a simple forward planner. Similarly, the application of informed heuristics for the ADFS domain is future work.

C. Example Application

To demonstrate the application of our approach, we provide a simple example in Figure 3 where the goal of the robotic agent is to design a structure to climb the object G using a subset of the six available objects (gray) on the left. Note that the maximum height the robot can climb up to, $yMax$, is 2 units, the height of object o_3 and the maximum distance, $xMax$, it can skip in climbing is 0.5 units.

In the figure below, we show our final output (blue) where the object o_4 is placed above o_5 right after o_6. Note that the red boxes represent the abstract design that only considers the state literals, but not the object sizes or masses. The final plan

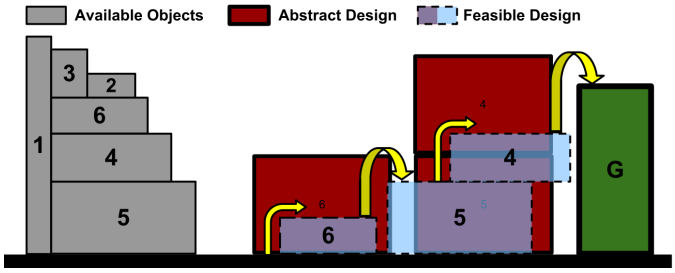


Fig. 3. The final result of a simple problem of obstacle climbing. The yellow 'right-turn's represent the actions 'WalkFromGround' and 'WalkAbove', and the 'u-turn's represent the 'Jump' actions.

is: {PutGround(o_6), WalkFromGround(o_6), PutGround(o_5), PutOn(o_4,o_5), Jump(o_6,o_5), WalkAbove(o_5, o_4), Jump(o_4,G)}.

We search for a path from the initial state s_0 ={At($ground$), Used(G), Unused(o_i), CanGo(o_i) $|1 \leq i \leq 6\}$ to a goal state s_g such that the literal At(G) $\in s_g$. Note that the literal CanGo(o_i) is removed once the robot jumps to an object o_i and used to avoid cycles in the search. Figure 4 presents the search tree of the backtracking forward planner and the outcome plan in the form of green directed edges. The tree has the following abbreviations to save space: (1) an action A(x:y,z) represents the group {A(x,z), A(x+1,z), ..., A(y,z)} leading to a set of 'unseen' states, (2) the dashed edges contain two actions with the top one applied first (the middle nodes are not expanded), and (3) the red state s_{35} has a tree of backtracks not shown.

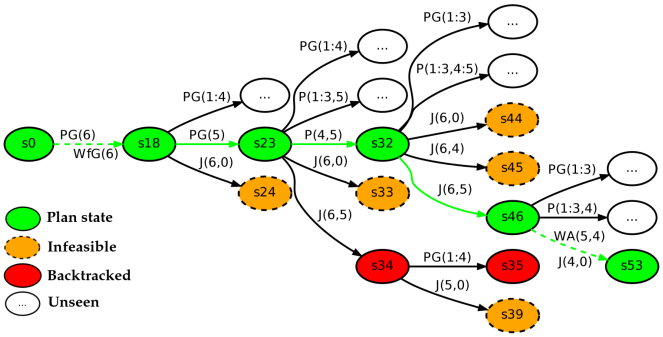


Fig. 4. The planning graph for the simple example where the green arrows represent the desired plan and the state $s53$ satisfies the goal requirements.

Several points need to be made about the forward planner tree depicted above. First, note that the children of a state are expanded bottom up. For instance, the children of the state s_{23} are analyzed in the order of $\{s_{34}, s_{33}, s_{32}\}$ where the last created child is analyzed first. Secondly, as a state is generated, the feasibility test is conducted for the partition of the configuration space it represents. If a state fails the test, such as one of the states s_{24}, s_{33} and s_{44}, it is useless to examine its children and thus, it is discarded. Lastly, the set of "generic" actions are first instantiated with the possible combinations of objects without a preference on the ordering of the objects. Then, the instantiated actions with satisfied preconditions are prioritized with the order {WalkFromGround, Jump, WalkBelow, WalkAbove, PutOn, PutGround}.

Starting from the initial state $s0$, the planner applies two actions, PutGround(6) and WalkFromGround(6), taking the robot on top of object o_6. Next, the planner skips the unfeasible state s_{24} and places o_5 on the ground, moving to state s_{23}. Following the action ordering, the planner goes to state s_{34}, prematurely jumping to o_5. At this point, it can neither reach the goal G because o_5 is too low nor climb upon other objects - thus, the backtracks in s_{35}. However, falling into this branch leads the planner to investigate approximately four thousand nodes, a common problem with depth first search that can be fixed with domain specific heuristics.

After *backtracking* from state s_{35} to state s_{23} and skipping the unfeasible state s_{33}, the planner makes the correct choice at each level. Before jumping to o_5, the robot places object o_4 on top of it (s_{32}). This way, after the jump, the robot walks above to object o_4 (s_{46}) and jumps to the goal object G (s_{53}). In the final state, the feasibility test solves for 7 equality constraints, 25 inequality constraints, and considers the mass of the robot at 4 locations (before and after the two jumps) to guarantee the stability of the structure as the robot traverses it.

D. Algorithm

Algorithm 1 demonstrates the complete planner for constraint space problems where the inputs are a domain definition $\mathbf{D}$, the list of desired goal literals $\mathbf{G}$ and an initial state s_0. In the following, we describe the details of the algorithm. Given that actions are parameterized by object names and represented with strings, the instantiate function at line 1 creates the set of all instantiated actions $\mathbf{A}(\mathbf{D})$, by replacing the object names with the parameter names of each action. At line 2, the stack of states to be inspected is created with the initial state.

Algorithm 1: ConstrainedForwardPlanner()

```
   Input: domain: objects properties and generic actions;
   Input: goals: list of goal literals to be fulfilled;
   Input: initialState: discrete literals;
   Result: configurations: a feasible value in goal subspace;
1  allActions ←instantiate(domain);
2  stateStack ←createStack(initialState);
3  while stateStack not empty do
4      state ← stateStack.pop();
5      actions ←stateActions(actions);
6      foreach action in the set actions do
7          child ←applyAction(state, action);
8          confs ←domain.checkFeasible(child);
9          if confs = ∅ then stateStack.push(child);
10         else if goals ⊂ child then return confs;
11 return ∅;
```

In the main loop between lines 3 to 10, at each iteration, a new state is pulled from the stack. For each state s_i, the set of available actions $A(s_i)$ is determined at line 5 by checking for the preconditions of each instantiated action: $A(s_i) := \{A_i \in \mathbf{A}(\mathbf{D}) | A(s_i).pres \subset s_i\}$. Once an action is chosen, a new state s_j is created from the parent s_i by applying the effects of the action A_i: $s_j := (s_i \setminus A_i.dels) \cup A_i.adds$. Once the child state is retrieved, a domain specific feasibility test is called (line 8). If the child is feasible and satisfies the goal literals, the feasible values are returned (line 10); otherwise, it is pushed to the stack and the loop continues (line 9).

The algorithm makes use of the action constraints in two ways. First, it evaluates the feasibility of seemingly plausible states. For instance, although the preconditions of the action PutOn(o_1,o_2) are satisfied in a state, it is not guaranteed that o_1 can be balanced on o_2 (e.g., lack of other objects to counterbalance). Secondly, the feasibility evaluation outputs feasible configurations that are the desired goals of the search.

Function FeasibilityTestADFS presents the feasibility test for the ADFS domain where 5 types of constraints are combined: simple and complex action constraints, balance, collision and problem constraints. All the constraints are optimized with the linear optimization toolbox named 'glpsol' [7]. In the following, we describe the types of constraints.

Function `FeasibilityTestADFS`

```
   Input: state: discrete literals that define constraints;
   Result: configurations: locations of the final design;
1  constraints ← state.equalities ∪ state.inequalities
2  foreach complexConst in the set state.complex do
3      constraints.add(apply(complexConst, state));
4  addBalanceConstraints(constraints);
5  addCollisionConstraints(constraints);
6  addProblemConstraints(constraints);
7  return glpsol(constraints);
```

Figure 2 contains examples of simple constraints defined with the literals 'Eq()' and 'InEq()'. A different complex class is created to induce constraints after an object is added to the system. For instance, the action 'Jump(o_1,o_2)' constraint has the complex constraint $leftSpace$ which guarantees that there are no objects above object o_2 for a horizontal distance $xMin$ from its left edge so that the robot has room to move. Such a constraint involves all the objects in the environment, making it necessary that it is considered after the action is taken.

The balance constraints for an object o_i induce two inequality constraints limiting the center of mass of all the objects above o_i to stay in between the edges of o_i (line 4). Let O_i be the set of objects o_j such that there exists a chain $\{On(o_j, x_1), On(x_1, x_2), ..., On(x_n, o_i)\}$, let $\hat{m}_i$ be their total mass and let $\hat{x}_i$ be their center of mass: $\sum_{o_j \in O_i} \frac{m_j}{\hat{m}_i} * x_j$. The two inequalities are as follows: '$x_i - \hat{x}_i \leq \pm 0.5 w_i$.'

To ensure stability, the location of the robot is taken into account in the balance constraints. We make the observation that the torque induced by the robot is maximum when the robot is at the edges of the block it is on - that is exactly before and after a jump. Thus, the effect of other actions, i.e. WalkAbove and WalkBelow, is limited by the Jump actions and if the structure is stable when the robot jumps, then it would be stable all through its path. Thus, to realize stability, for each action Jump(o_1,o_2), we simulate torques in the

described balance equations at the locations $x_j^1 = x_1 + 0.5w_1$ and $x_j^2 = x_2 - 0.5w_2$ with the robot mass m_r.

The collision constraints (at line 5) guarantee that objects do not overlap each other in their final position. To implement this, the height of each object is determined using the $On(x, y)$ constraints and for the objects o_1 and o_2 that might collide, the inequality constraint, '$x_2 - x_1 \geq 0.5(w_1 + w_2)$', is added.

At line 6, the problem constraints are introduced to the system of constraints. Although these constraints vary based on the problem, for problems with an obstacle G, the location of the obstacle has to be fixed using equality constraints. For the fire example in Figure 1, none of the objects were allowed to be within the region with the following set of inequality constraints: $\{x_i + 0.5w_i \leq fire_x, y_i - 0.5h_i \geq fire_y | \forall o_i \in O\}$.

V. Evaluation and Future Work

Figure 5 demonstrates an example where the goal is to climb a tall obstacle. The properties of the objects in the environment, their small sizes, push the planner to use a large number of objects in the design. In this section, we use this figure as a motivating example to evaluate the complexity of our proposed algorithm and its limitations.

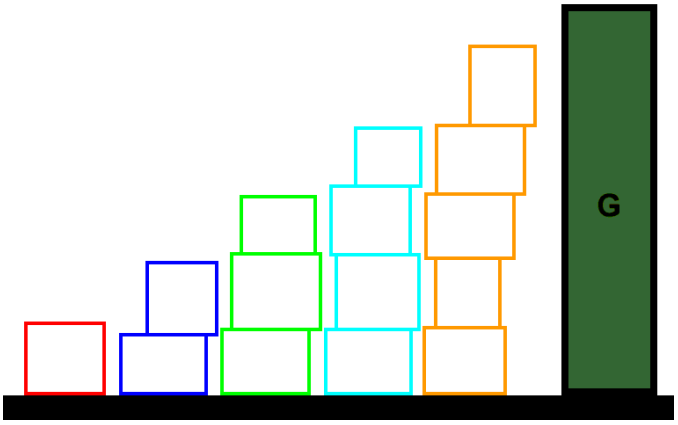


Fig. 5. An example of a functional structure where the goal is to climb a tall obstacle using a large number of smaller objects.

For a problem domain with n objects and m generic actions, the search tree at worst has mn^2 instantiated actions assuming an action affects at most two objects. For a maximum plan length of d and a branching factor of mn^2, the worst case complexity of the depth-first search is $O(mn^{2d})$. Such a complexity is expected given the simplicity of the search method and complexity can be improved extensively by using informed heuristics. For the fire example with 5 objects and 6 actions, the search expands 4952 nodes and takes an average of 12.98 seconds (10 trials) on a 4.5GHz Intel i7 processor.

In addition to the complexity of the search in terms of nodes, the average duration of a feasibility test also plays an important role in overall efficiency. The relaxation we have made in allowing only linear equality and inequality constraints allows for the use of efficient convex optimization methods. For instance, for the final configuration of the example in Figure 5 with 15 equality and 59 inequality constraints, the feasibility test takes 4 milliseconds. In generalizing our approach to quadratic and general non-convex constraints, it will become imperative to find similarly efficient feasibility methods.

Lastly, we address the issue of buildability of the designed structures. In this work, we make the assumption that the agent has access to the objects when it wants to place them in the world or that once a plan is made, it can build the structures, especially those that require counterbalancing, off site and move them to correct positions before utilizing them. It is clear that future work has to consider the problem of buildability to realize the autonomous construction of automated designs.

VI. Conclusion

In this work, we introduce the automated design of functional structures (ADFS) and identify the broader class of problems it belongs to: problems with high dimensional continuous spaces that can be partitioned into subproblems by semantically meaningful actions. We observe that actions with discrete propositions and continuous constraints effectively partition the space of solutions. Thus, we reformulate the problem where the goal is to plan in the space of constraints, adding constraints until a desired goal subspace is shaped.

Following the presentation of a constraint space planner, we provide a detailed formulation of the ADFS problem and demonstrate an example application of our approach. Moreover, we prove that the constraint space planner is complete given that the feasibility module is sound. We discuss the computational limitations regarding the space of candidate designs and the merits of convex constraints in efficiency.

In conclusion, we believe this work presents a principled treatment of problems in high dimensional continuous spaces using classical planning approaches and hope to investigate a wider range of problems in future work, in the domains of automated design and construction, towards full autonomy.

Acknowledgements

This work was supported by ONR Grant #N000141210143: Autonomous Discovery of Object Properties: Robots that Create Simple Machines.

References

[1] B. Akgun and M. Stilman. Sampling heuristics for optimal motion planning in high dimensions. In *IROS*. IEEE, 2011.
[2] M. Buss, H. Hashimoto, and J.B. Moore. Dextrous hand grasping force optimization. *Robotics and Automation, IEEE Transactions on*, 1996.
[3] U. Dorndorf, E. Pesch, and T. Phan-Huy. A time-oriented branch-and-bound algorithm for resource-constrained project scheduling with generalised precedence constraints. *Management Science*, 2000.
[4] M. Kalisiak and M. van de Panne. RRT-blossom: RRT with a local flood-fill behavior. In *ICRA 2006*, pages 1237–1242. IEEE, 2006.
[5] O. Khatib and J. Burdick. Optimization of dynamics in manipulator design: The operational space formulation. *IJRA*, 1987.
[6] J.J. Kuffner Jr and S.M. LaValle. RRT-connect: An efficient approach to single-query path planning. In *ICRA*. IEEE, 2000.
[7] A. Makhorin. Modeling language GNU Mathprog. In *Structure*, 2007.
[8] S.J. Russell and P. Norvig. *Artificial intelligence: A Modern Approach*. Prentice Hall, 2010.
[9] M. Stilman. Task constrained motion planning in robot joint space. In *IROS*. IEEE, 2007.
[10] V. Vidal and H. Geffner. Branching and pruning: An optimal temporal pocl planner based on constraint programming. *Artificial Intelligence*, 2006.