LEARNING ABSTRACTIONS FOR PLANNING

by

BRIAN COOK

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT ARLINGTON

May 2020

To my parents, Lynn and Judy, for instilling in me a love of learning and for supporting my education even when it led me far away.

ACKNOWLEDGEMENTS

I would like to thank my advisor Manfred Huber. His insight and creativity have constantly challenged and inspired my thinking, and I am grateful for his advice and support over the past 7 years. I also wish to thank my committee members (past and present) Gergely Zaruba, Farhad Kamangar, Gian-Luca Mariottini, Taylor Johnson, and Bill Beksi. I appreciate their interest in my research and for investing their time to serve in my dissertation committee.

I thank my high-school teacher Martha Pearson for encouraging me to think big and inspiring me to pursue graduate studies.

Most of all, I am grateful for my wife Tamara. Her unwavering love, patience, support and encouragement made this work possible.

May 29, 2020

ABSTRACT

LEARNING ABSTRACTIONS FOR PLANNING

BRIAN COOK, Ph.D.

The University of Texas at Arlington, 2020

Supervising Professor: Manfred Huber

Planners for hard problems must exploit domain-specific structure to find solutions efficiently. Yet, hand-engineered solutions and optimizations are often expensive and difficult or impossible to adapt to other problems. This work applies automatic machine learning techniques to increase planner performance for specific problem domains and to learn useful abstract representations for planning. In particular, this dissertation develops methods to address important aspects of learning in planning in four different areas:

State-of-the-art domain-independent classical planners utilize multiple search heuristics and decide how to allocate computational effort between heuristics prior to planning. This work presents a heuristic planning algorithm that uses a learned model of heuristic search dynamics to dynamically allocate computational effort to available heuristics during planning.

Non-parametric function approximators can be used to represent state transition models, reachability estimators and distance functions for planning. However when training data is not uniformly distributed, variations in sample density can result in local neighborhood bias that negatively affects accuracy. Two new algorithms are presented for k-nearest neighbor (kNN) regression and classification that explicitly compensate for the asymmetric distribution of local neighborhood samples.

A novel abstraction-guided planning algorithm using control policies is presented and implemented for a multi-goal physics-based game. This work develops learned predictive models for abstract state connectivity and control policy utility and shows they can increase the efficiency of exploration and construction of the abstract model.

Lastly, this work proposes a novel approach for automatically identifying useful abstract states using random transition sampling and graph analysis. Experiments show the method yields results similar to hand-engineered abstractions implemented by human experts for the same domain.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF TABLES

CHAPTER 1

OVERVIEW

1.1 Introduction

The field of *planning* encompasses an enormous range of problems encountered in robotics, artificial intelligence (AI), control theory, and other disciplines. While incredibly diverse, these problems share certain fundamental characteristics. Namely, some system in the world is represented by a *model*, in which the *state* of the system evolves over time in response to a sequence of *actions* taken by an agent. A planning problem is then defined by such a model along with an initial state and a set of *goals*. The task of a planner is to find a sequence of actions that best achieve the goals. This sequential decision-making is the defining quality of planning problems.

Important and challenging planning problems abound, including game-playing, autonomous vehicle and spacecraft control, logistics, manufacturing assembly, drug design, and industrial control. Solving these problems efficiently is of immense value. A famous example is the DART planner used by U.S. forces during the Persian Gulf conflict in 1991 for logistics planning and scheduling. The Defense Advanced Research Project Agency (DARPA) stated the savings in time and resources by this one application more than paid back the preceeding 30 years of investment in AI by DARPA. [14]

Yet while planning is important, it is hard. Complexity results vary depending on the problem representation and constraints, but many problems of interest are either PSPACE-complete or EXPSPACE-complete for finding feasible plans. When searching for optimal plans, many become NEXPTIME-complete. [24]

Nonetheless, automated planners have achieved considerable success by exploiting domain-specific structure to find solutions efficiently. These domain-specific planners are typically hand-engineered and optimized for a specific class of problems, often with a customized domain-specific representation. While they may be effective, such solutions also tend to be expensive and rigid, and may be difficult or impossible to adapt to other problems.

Domain-independent planners on the other hand are designed to address a broader range of problem domains and thereby reduce the engineering cost of solving particular problems by minimizing domain-specific components. This increased flexibility usually comes at the cost of efficiency since such a planner can make no prior assumptions about constraints or patterns that might be exploited for a particular problem domain.

In the real world, a planner is usually dedicated to solving many similar problems. For example, a pick-and-place robot in a warehouse might be presented with thousands of distinct tasks, but the types of items, the warehouse environment, and capabilities of the robot are similar for each task.

These observations raise a basic question. Can a planner designed for a broad set of problem domains learn about a particular problem domain and then exploit that knowledge to improve its performance when solving new problems from that domain?

The term *learning* itself is open to a range of definitions, but for our purposes we adopt the view that a system that improves performance with experience is said to be learning.

The past 25 years have witnessed enormous progress in machine learning. Learned function approximators based on neural networks and support vector machines have demonstrated impressive success in a wide variety of applications, including detecting and classifying objects in images, speech recognition and translation, identification of biological markers, and many more [38].

However, these technologies have thus far seen only limited application to planning. The aim of this research is to investigate applications of machine learning to improve the efficiency and quality of planning on both symbolic and continuous problem domains.

1.2 Statement of Contributions

The first contribution is a novel approach for planning based on heuristic search, in which the planner decides how to allocate computational effort based on estimates of search progress. The DH1 algorithm for dynamically selecting heuristic planners is introduced. Given a set of training problems drawn from a target problem distribution, and given a base set of domain-independent heuristic planners, a predictive model is trained to estimate heuristic progress on problems in the domain. After training, these estimates are then used to dynamically select the most promising heuristic during the planning process. This is the first work that uses a learned model of the heuristic search dynamics to dynamically select a heuristic or planner during planning. Experimental results on planning benchmark problems show that dynamic heuristic selection using the learned model can solve more problems than strategies that statically allocate processing time, one of which allocates all time to the single best heuristic for a domain, or another that equally allocates time to each base heuristic.

A second contribution is a pair of new algorithms for k-nearest neighbor (kNN) regression and classification. Non-parametric function approximators are commonly used to represent models used in planning, including state transition models, reachability estimators and distance functions. However, when the training data for such a model is not uniformly distributed, as is usually the case, variations in sample density can result in *local neighborhood bias* that negatively affects the accuracy of kNN-based estimators. This work introduces two novel kNN algorithms that adjust the weights of the k-nearest neighbors to balance the influence of samples from opposing regions of space. This is the first approach to explicitly compensate for the asymmetric distribution of local neighborhood samples. Experimental results on synthetic and real-world data show that these algorithms improve accuracy under a defined range of conditions.

A third contribution is an abstraction-guided kinodynamic planning algorithm for multi-goal problems using control policies. The algorithm is implemented in a competitive planner for the Geometry Friends (GF) game. GF is a challenging physics-based 2-dimensional problem-solving game featured in several recent competitions. [51] When presented with a problem instance, the planner builds an abstract model of the state space by identifying regions where the game agent is stable and controllable. Sampling and simulated rollouts are used to estimate connectivity between the abstract regions, then graph search (A*) is used to find a candidate abstract plan that achieves the high-level goals for the problem. The high-level plan guides the agent and directs the lower-level control policies responsible for reaching a target location and velocity within a local region. This planner is able to find feasible and high-quality solutions for all of the published GF problem instances including those from past competitions.

A fourth contribution is a method for using predictive models to reduce the simulation effort required to explore a state space and construct an abstract model. The models predict whether particular control policies are likely to be useful in a range of situations. Using these predictions the planner can avoid allocating simulation effort for policies that are unlikely to improve the abstract model. Models are developed and evaluated for the GF planning domain and demonstrate the predictive method is effective in finding higher-quality plans with less simulation effort than comparable methods without prediction.

The final contribution is a novel sampling-based approach for learning abstract states. For continuous state spaces the task of identifying abstract states for use by higher-level symbolic planning often requires hand-engineered domain-specific abstractions. This work investigates whether it is possible to identify useful state abstractions using only random sampling and a minimal set of first principles. The proposed approach uses random transition sampling and graph analysis to identify strongly-connected regions of space that are good candidates for abstract states. Using the identified regions, we then seek to learn an agent control policy to enable the agent to reliably transition between any pair of states in an abstract region. The approach is applied to problem instances from the GF planning domain and successfully identifies abstract regions similar to the hand-engineered abstractions used by every competitive GF planner. An agent control policy is learned for the identified regions using a Deep-Q Network and shows promising results.

1.3 Outline

Chapter 2 reviews state-of-the-art planning approaches for symbolic and continuous domains along with learning methods. Chapter 3 introduces the DH1 algorithm for dynamically allocating search effort to heuristics. Chapter 4 describes two new algorithms for k-nearest neighbor (kNN) regression and classification that can improve accuracy by compensating for non-uniform training sample distribution in the local neighborhood. Chapter 5 introduces the AGPlan and AGAgent algorithms for abstraction-guided planning and execution. Chapter 6 develops predictive models for estimating whether control policies are likely to be useful in particular situations and applies the predictions to reduce unproductive simulation effort. Chapter 7 proposes a novel sampling-based approach for identifying abstract states that may be useful for symbolic planning. A policy is then learned for state-to-state control within the identified regions.

CHAPTER 2

BACKGROUND AND RELATED WORK

2.1 General Planning

Planning is concerned with selecting a series of actions that affect the state of a system to achieve some set of goals, possibly subject to defined constraints. Systems in planning may be formally modeled as *state-transition systems* [24], that are described as a tuple $\Sigma = (S, A, E, \gamma)$ where:

- $S = \{s_1, s_2, ...\}$ is a finite or recursively enumerable set of states
- $A = \{a_1, a_2, ...\}$ is a finite or recursively enumerable set of actions that can be selected by the planner. Actions are sometimes also referred to as *operators*.
- $E = \{e_1, e_2, ...\}$ is a finite or recursively enumerable set of possible events that are externally triggered and not under control of the planner
- $\gamma = S \times A \times E \mapsto 2^S$ is a state transition function

A system is considered *static* if the set of events E is empty. In this case the system has no dynamics and the state changes only when the planner selects an action.

A system is *deterministic* if for every state s, action a and event e, $|\gamma(s, a, e)| \leq 1$, i.e. there is at most one possible outcome state for any action and event occurring in a state.

A system is *fully observable* if the planner has complete access to the current state s. In a *partially observable* system the planner instead is provided an observation function $\eta : S \mapsto O$ where O is a set of observation values.

A planning problem is then specified by a system Σ, an initial state s_0, a set of constraints, and a set of objectives.

Constraints on solutions may take many forms, but a common constraint specifies which states of the system are allowed in a valid plan, $S \mapsto \{true, false\}$. For example, a predicate function $isCollision(s)$ that indicates whether the system is in a disallowed collision state.

Objectives also may be defined in a variety of ways. Objectives are commonly specified as one of the following:

- A set of goal states G with the objective of finding a sequence of actions and states leading to a goal state.
- A reward function $R : S \times A \mapsto \mathbb{R}$ that assigns value to actions taken in different states, with the objective of finding a sequence of actions and states that maximize some function of the reward.

2.2 Classical Planning

Classical planning is concerned with planning problems that are fully-observable and deterministic, with a finite number of states and a specified set of goal states. Actions and events are instantaneous without explicit representation of time.

The earliest work on classical planning specified problems using the well-known STRIPS propositional representation [20]. Recent work typically uses the more compact PDDL (planning domain definition language) which extends STRIPS with first-order literals and logical connectives to specify operator preconditions and conditional effects, as well as the ability to define axioms [39]. An alternative formulation is the *multi-valued planning task (MPT)* that instead represents each state as a tuple of variables, each of which has a discrete, finite set of values [27]. All of these representations have equivalent expressive power and problems may be translated between them; however PDDL and MPTs have the advantage of explicitly representing some structure that is only implicit in a grounded propositional representation.

2.2.1 Heuristic Search

A classical planning problem implicitly defines a finite directed graph rooted at the initial state and thus standard graph search algorithms may be used to search for solutions. A commonly used search algorithm is A* as shown in Algorithm 1.

A* is an informed search algorithm that makes use of a heuristic estimate h of the *cost-to-go*, which is the optimal cost from the current state to the nearest goal state. Nodes are selected for expansion from the open list according to the f-value, which for A* is $f(n) = g(n) + h(n)$ where g is the cost to reach node n from the initial state and h is the heuristic estimate for n.

If a heuristic estimate h is always less than or equal to the true cost-to-go, then the heuristic is said to be *admissible*. A* search finds optimal-cost solutions when using an admissible heuristic.

Heuristic functions vary in how informative they are. The least-informative heuristic is $h(n) = 0$, which is admissible and results in optimal solutions by A* (assuming non-negative action costs). In this case, however, nodes in the open list are ordered entirely by the cost-to-reach $g(n)$ resulting in an uninformed breadth-first search that evaluates every possible solution with cost less than an optimal solution.

At the other extreme, A* with a perfect heuristic function with $h(n)$ equal to the true cost-to-go will result in only expanding nodes along an optimal path to a solution.

A* balances $g(n)$ and $h(n)$ equally, choosing to explore nodes with the lowest total estimated cost. Greedy best-first search (GBFS) instead uses $f(n) = h(n)$ and considers only the heuristic estimated cost-to-go with the hope of finding a solution more quickly although losing any guarantees of optimality.

Heuristics are limited in their applicability. A customized domain-specific heuristic may be highly informative for a class of problems, but this superior performance for some problems comes at the cost of poor performance on others. The No Free Lunch theorem [63] shows that for general heuristic search, no heuristic is superior to any other when considering all possible search problems. The value added by a heuristic is inherently tied to the assumed structure of the problems it is designed to solve. Even so-called domain-independent heuristics rely heavily on implicit assumptions regarding the problems they solve.

As a result, a variety of multi-heuristic search approaches have also been investigated. For example, a search agent may utilize multiple open lists, each organized by a different heuristic, and select one node at a time from each open list in a round-robin manner. Another approach is to assign the heuristic value for a node by taking the minimum, maximum, or average of a set of independent heuristic estimates. A fixed multi-heuristic strategy is effectively another heuristic. Adding heuristics may improve or degrade performance depending on the quality of the heuristics.

2.2.2 Heuristic Planning

The first successful demonstration of heuristic state-space search for large classical planning problems was the HSP planner that won the AIPS98 international planning competition [6]. Since then, heuristic planners have come to dominate classical planning competitions, using a variety of increasingly sophisticated domain-independent heuristics.

Yet competition results also confirm that no one heuristic planner achieves superior performance on every problem domain. A planner that out-performs on a

Algorithm 1 A* Search

```
1: input: search problem $problem$
2: output: a solution, or failure
3: $openList \leftarrow$ empty priority queue
4: $closedList \leftarrow \varnothing$
5: $openList \leftarrow Insert(openList, problem.InitialState, 0)$
6: while not $Empty(openList)$ do
7:     $node \leftarrow Pop(openList)$
8:     if $problem.IsGoal(node.State)$ then
9:         return $Solution(node)$
10:    $closedList \leftarrow Add(closedList, node.State)$
11:    for $a \in problem.Actions(node.State)$ do
12:        $child \leftarrow ChildNode(problem, node, action)$
13:        $g \leftarrow child.Cost$
14:        $h \leftarrow CalculateHeuristic(problem, child.State)$
15:        $f \leftarrow g + h$
16:        if $child.State \notin closedList$ then
17:            $openList \leftarrow Insert(openList, child.State, f)$
18: return $failure$
```

majority of domains may still exhibit very poor performance on others. Likewise, a planner that is quite poor overall may nonetheless excel at problems in a particular domain.

Consequently the most competitive classical planners today are those that select one or more of the most promising planners to apply to a particular problem.

Portfolio planning allocates processing time to multiple independent planners in the hope that one of them solves the problem.

Fast Downward Stone Soup (FDSS) [29] is a static generic portfolio of base planners selected in advance based on each planner's performance on all prior competition domains and problems. FDSS proved highly competitive at IPC 2011, winning the sequential optimizing track and placing second in the sequential satisficing track.

Other planners learn static domain-specific portfolios. For example, PbP2 [23] uses learned knowledge from a set of training problems to pre-select and parametrize a portfolio of promising base planners for a domain. Using this approach PbP2 won the learning track at IPC 2011.

IbaCop2 [9] is a problem-specific portfolio planner that uses a learned classifier to select 5 base planners for each problem. The classifier uses features extracted from a static analysis of the problem prior to planning, and the available processor time is divided equally among the selected base planners. Using this approach IbaCop2 won the sequential satisficing track of IPC 2014.

All of these portfolio planners attempt to select the base planners that are most likely to solve a problem, but they treat them as black boxes. The internal dynamics of the base planners are not considered during the planning search process.

Other work has been done in the area of dynamic heuristic selection during search.

The Fast Downward planner [27] introduced the use of multiple heuristics with a separate open list for each heuristic as *multi-heuristic best-first search* using the causal graph and FF heuristics. In addition it maintains separate open lists restricted to nodes reachable by preferred operators identified by each heuristic. The planner alternates between these open lists when selecting the next node for expansion. Successor nodes are added to all of the open lists. This strategy improved overall performance resulting in Fast Downward winning the classical planning track at ICAPS 2004.

Alternating open lists were also used by the well-known LAMA 2008 and 2011 planners [52]. In addition to a new landmark heuristic, LAMA added a boost in priority for heuristics that make progress by achieving a new minimum h-value. LAMA performed very well, winning the sequential track at IPC 2008.

In addition to alternation, other multi-heuristic combination strategies have been investigated such as maximum, sum, tie-breaking, and pareto dominance [54]. Yet in all of these cases, the combination strategy for the heuristics is pre-selected and does not take into account the current planning state. Heuristic selection is determined by a hand-engineered fixed policy that is the same for every domain and problem.

Two approaches to the problem of dynamic heuristic selection are presented in [48]. The first approach, Meta-A*, selects the heuristic at each step that has the lowest total g-value plus weighted h-value, thereby favoring heuristics that appear closer to a solution. The second approach treats heuristic selection as a multi-armed bandit problem where heuristics are given reward when they achieve a new minimum h-value. Both approaches dynamically adapt to prefer heuristics that are making progress, but the determination of progress is effectively limited to the rate of decline in the minimum h-value. No learned knowledge is applied across problems.

For optimal planning, *selective max* uses a problem-specific learned classifier to dynamically select one heuristic from a set of admissible heuristics to evaluate for each state. This approach is shown to outperform any one heuristic or taking the max of the heuristics [17]. However, the classifier utilizes only elementary features corresponding to the state variables of the particular planning problem and cannot be applied across problem instances.

RA* approaches multiple-heuristic selection for optimal planning as a utility problem [4]. Given a set of admissible heuristics for a problem, RA* initially randomly samples frontier nodes during the search in order to estimate the relative strength of each heuristic as well as the average per-node generation and evaluation time. Information from the sampling period is used to select a combination of heuristics with the highest observed utility. The remainder of the search evaluates that combination of heuristics on every node and takes the max as the h-value. Heuristic selection is static after the initial sampling period. Again, no learned knowledge is applied across problems.

None of the above approaches consider the *dynamics* of a heuristic search. During a search, the sequence of node values consumed from the open list(s) can be treated as a time-series signal. When a search is stuck in a large plateau the values may change very little, while in other regions the values may oscillate as branches of the search graph are explored. A natural question investigated by this work is whether the behavior of a heuristic signal over a span of time can provide useful information for predicting the utility of that heuristic in real-time for a particular problem.

2.3 Continuous Planning

While a large number of important real-world problems can be addressed using classical planning, continuous problem domains require a very different approach. Examples include robot arm and manipulator motion planning, autonomous vehicle path planning, chemical industrial plant control, and protein folding analysis.

2.3.1 Kinematic Planning

In the terminology of continuous planning, the *workspace* refers to the 3D space in which the system itself is located. The *configuration space* (C-space) refers to the set of possible transformations that can be applied to a robot, usually represented as a real-valued vector $q \in \mathbb{R}^n$. For example, the configuration of a simple differential-drive robot on a flat surface may be represented as $q = [p_0, p_1, \theta]$ where p_0 and p_1 specify the 2D position of the robot and θ the orientation. C-space is partitioned into two regions, free space C_{free} and obstacle space C_{obs}. *Kinematic planning* is restricted to the configuration space and searches for collision-free paths that are geometrically feasible but does not consider system dynamics, i.e. velocity or acceleration. This is useful for a wide range of problems, however such a kinematic solution may not be dynamically feasible in general.

For kinematic planning problems the use of graph search methods on discretized representations is typically too computationally expensive due to the high dimensionality of the search spaces. While a wide variety of alternative methods such as potential fields with random walks have also been explored, randomized sampling-based methods have emerged as the preferred approach for many applications. They can often provide fast suboptimal solutions to challenging high-dimensional problems.

The most common algorithms are based on the Probabilistic Roadmap Method (PRM) [34] and Rapidly-exploring Random Trees (RRT) [36].

PRM is a multi-query planning algorithm designed to solve multiple problem instances in the same environment. During a learning phase, random samples are selected from the obstacle-free region of the configuration space and are connected to form a roadmap. For each problem instance, the starting and goal states are connected to the roadmap in order to find a path.

Roadmaps sample space uniformly and are not goal-directed. They are designed to be reusable in a static environment. For single-query planning they may not be efficient since they may spend much of their time exploring space that is either not reachable from the initial state or is not relevant to finding the goal.

The RRT algorithm is designed for single-query planners. Instead of creating a forest of connected components as with PRM, RRT incrementally builds a tree expanding outward from the starting configuration toward the goal. The general RRT algorithm is shown in Algorithm 2.

RRT initializes a tree starting at the initial configuration. It then explores the configuration space by selecting samples from a dense random or quasi-random sequence and attempting to connect the tree to each sample. Goal-seeking is achieved

Algorithm 2 RRT

```
1: function query($q_{init}$, $q_{goal}$)
2: $T \leftarrow$ newTree($q_{init}$)
3: while not $timeout$ do
4:     $q_{target} \leftarrow$ getNextSample()
5:     $q_{new} \leftarrow$ growTree($T, q_{target}$)
6:     if $q_{new}$ and $\rho(q_{new}, q_{goal}) < \epsilon$ then
7:         return extractSolution($q_{new}$)
8: return $failure$

1: function growTree($T, q_{target}$)
2: $q_{nearest} \leftarrow$ nearestNeighbor($T, q_{target}$)
3: $u_{best} \leftarrow$ selectControl($q_{nearest}, q_{target}$)
4: if $u_{best}$ then
5:     $T \leftarrow$ T + newEdge($q_{nearest}, u_{best}$)
6:     $q_{new} \leftarrow$ simulate($q_{nearest}, u_{best}$)
7: return $q_{new}$

1: function selectControl($q, q_{target}$)
2: $d_{min}, u_{best} \leftarrow \rho(q, q_{target}), \varnothing$
3: for $u \in U$ do
4:     $q_{new} \leftarrow$ simulate($q, u$)
5:     if $q_{new}$ then
6:         $d \leftarrow \rho(q_{new}, q_{target})$
7:         if $d < d_{min}$ then
8:             $d_{min}, u_{best} \leftarrow d, u$
9: return $u_{best}$
```

by including a goal configuration in the sample sequence some fraction of the time. Compared with PRM, RRT has the advantage that the trees only include configurations that are reachable from the initial point, so it spends less time considering areas of space that are unreachable and irrelevant to the solution.

The behavior of RRT is dependent on a number of key elements that may be modified to create variants of the algorithm:

- *Sampling Strategy*: The random sample sequence is the core of RRT. It typically uses a dense uniform random or quasi-random sequence of configurations, usually but not always restricted to obstacle-free configuration space. The frequency of goals appearing in the sequence controls how greedy the search is. Unbiased sampling assures uniform coverage of the space, but can be very slow to explore narrow passages. To compensate for this, some approaches bias the sampling toward bottlenecks between obstacles in the C-space.

- *Distance*: The distance function $\rho(q_a, q_b)$ estimates the cost of getting from q_a to q_b. In simple implementations ρ is often specified as a straight-line Euclidean distance $\rho = \|q_a - q_b\|$ but the true distance is often much more complex due to kinematic and dynamic constraints as well as obstacles in the environment. The `nearestNeighbor` and `selectControl` functions depend directly on ρ. When ρ is not accurate, RRT degrades badly as it fails to correctly select nearest neighbors and instead selects nodes that are further away and more difficult to connect to the target q_{target}. It may also generate control sequences that do not steer toward targets q_{target}.

  Distance function estimates typically ignore obstacles and thus tend to underestimate the true distance.

- *Local Steering*: The `selectControl` function is tasked with finding a control sequence to get from q to q_{target}. It effectively represents a local planner for a subproblem. In some cases it may be as difficult to solve this problem as the original planning problem.

  The pseudocode in Algorithm 2 shows an implementation which assumes the controls in U are discretized with fixed timesteps. Each step greedily selects the control value u that results in the greatest decrease in distance to the target. Like many steering methods, this approach is very sensitive to the distance function ρ.

- *Parent Node Selection*: The `growTree` function shown in Algorithm 2 always selects the nearest neighbor of the sample target as the parent node to extend toward the target, based on the distance function ρ. This may not always be the best choice, particularly if ρ is inaccurate. The planner may acquire and utilize additional knowledge to avoid selecting parent nodes that have little chance of successfully reaching the target.

RRT-Connect is a variation that allows multiple time-steps when attempting to extend the tree to a target and can span spaces faster. For each sample target it calls `growTree` repeatedly until q_{new} stops approaching q_{target}.

Algorithm 3 Bidirectional RRT

```
1:  function query($q_{init}$, $q_{goal}$)
2:  $T_a, T_b \leftarrow$ newTree($q_{init}$),newTree($q_{goal}$)
3:  while not timeout do
4:      $q_{target} \leftarrow$ getNextSample()
5:      $q_a \leftarrow$ growTree($T_a, q_{target}$)
6:      if $q_a$ then
7:          $q_b \leftarrow$ growTree($T_b, q_a$)
8:          if $q_b$ and $\rho(q_a, q_b) < \epsilon$ then
9:              return extractSolution($q_a, q_b$)
10:     $T_a, T_b \leftarrow T_b, T_a$
11: return failure
```

Much better performance is often achieved using bidirectional RRT as shown in Algorithm 3. This method utilizes two trees, one grown forward from the initial configuration $q_{initial}$, and a second grown backward from q_{goal}. The search alternates between the two trees. At each step, when a node q_{new} is added to tree T_a, an attempt is made to connect with the nearest neighbor in the other tree T_b. Since each tree provides a much larger target area than the singular q_{goal} there are many more opportunities to connect.

Standard RRT approaches can be fast but are proven to produce sub-optimal trajectories [42]. The RRT* algorithm modifies RRT to approach optimal solutions by rewiring the tree as it goes [33].

2.3.2 Kinodynamic Planning

The *state space* specifies all of the information describing a state of the system. For motion planning, this extends the configuration space with velocities and can be represented as $x = (q, \dot{q})$. In the differential drive robot example, the state space is thus 6-dimensional with $x = [p_0, p_1, \theta, \dot{p_0}, \dot{p_1}, \dot{\theta}]$.

Actions in continuous planning are typically specified by a control input vector $u \in U$ applied for some time interval. The full system transition equation is then defined as $\dot{x} = f(x, u)$ where f specifies both the kinematic and dynamic constraints of the system. *Kinodynamic planning* refers to planning in the full state space with kinematic and dynamic constraints.

State space planning introduces additional concepts. As with C-space planning, X_{free} and X_{obs} partition the state space into regions that are either in collision or not. In addition, the *region of inevitable collision* or X_{ric} specifies the set of states in X_{free} from which a collision is inevitable as a result of drift, regardless of what control input is applied. Conversely, states in X_{free} are called *viable* if control can avoid collisions, and the set of viable states $X_{viable} = X_{free} \setminus X_{ric}$ is sometimes called the *viability kernel.*

The *reachable set* from a state x_0 is defined as all states that are visited by any trajectories starting at x_0 for any sequence of control inputs $u \in U$. Cost-limited reachable sets, as the term suggests, are all states that can be reached within some cost limit.

Historically path planning and dynamic constraints have been decoupled. After generating a valid kinematic path, trajectory modification can be used to incrementally adjust the path to satisfy the dynamic constraints [35]. However, paths generated without regard for the system dynamics may be highly suboptimal and involve difficult maneuvers, and may be impossible to modify so that dynamic constraints are satisfied. As a result we are motivated to consider kinodynamic planning directly in the state space. State space planning is more difficult but can be shown to improve robustness, speed and energy efficiency of robots [62].

Kinodynamic planning is harder than path planning for a number of reasons:

- *High dimensionality.* The introduction of velocities essentially doubles the dimensionality of the search space.
- *Drift.* System dynamics usually preclude instantaneous changes in velocity. The resulting momentum constrains controllability and connectivity between states.
- *Distance.* The true distance function $\rho(x_a, x_b)$ is highly nonlinear and asymmetric and depends on the system transition equation. Obtaining an exact value requires solving an optimal motion planning problem and hence is not efficient or practical. For many problems estimating distance is computationally expensive and domain-specific. The true distance function is itself a function of the local steering controller behavior and of the obstacles in the workspace. As previously mentioned, many planning approaches, including RRT, rely on a reasonably accurate distance function and perform very poorly without one.

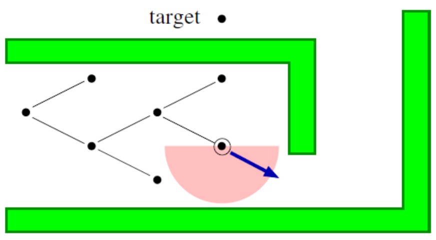
RRT tends to get stuck and explore poorly in kinodynamic problems. As a result numerous variants have been developed to address some of the issues.

RRT with Collision Tendency checking (RRT-CT) partially addresses the problem of RRT selecting the same nodes repeatedly [10]. Statistics are maintained for each node to keep track of how often a node connection attempt fails with a collision, and nodes with high rates of collision are less likely to be selected.

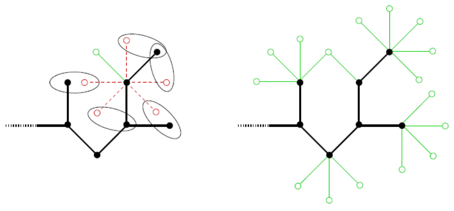
RRT-Blossom introduces a distinction between regressing and receding edges [31]. An edge is considered *receding* if it ends further from the target node than when it started. Indiscriminately adding receding edges can lead to wasted effort in areas already explored, and for this reason they are discarded by some algorithms. However, in some cases they are useful as shown in Figure 2.1. An edge is said to be *regressing* if it ends closer to an existing node in the tree than to the parent node. By filtering out regressing edges the search is able to explore faster, but one complication is keeping track of which nodes are non-viable so that they do not "block" other receding expansions.

Traditional planners consider the state of the agent and the environment separately, using them together only for collision checking. An alternative is to guide or steer local node expansion based on an egocentric representation of the local environment using local sensing. Define sensing functions σ_i that extract scalar sensing features from state x. A concatenated vector of sensor values is a sensory state s. The *locally situated state* of an agent combines the sensory state and local state, $\lambda = (s, \hat{x})$, where $\hat{x}$ excludes the global position and orientation of the agent.

Local sensing has been used to learn a viability model for avoiding non-viable regions [32]. Non-viable regions are problematic for sampling based planners since all attempts to connect graph nodes from X_{ric} will fail, and thus considerable effort is wasted if such nodes are selected frequently. To avoid this, a viability classifier $\Omega_v(\lambda) : \Lambda \mapsto \{\texttt{viable}, \texttt{nonviable}\}$ is trained to predict whether a given locally situated state λ is likely to be viable or not. It is then simple to modify existing algorithms by replacing checking for $isCollision(x)$ with $isCollision(x) \vee \neg isViable(x)$. Note that in



(a) A Useful Receding Edge



(b) Non-regressing Edges in Green

Figure 2.1: Receding vs. Regressing Edges (from [31])

a dual-tree environment, a second backwards classifier must be trained using reversed sensors.

2.4 Learning Approaches

Recent work has begun to explore applications of learning to improve planning.

Both RRT and PRM-based methods connect randomly sampled points to nearest nodes in a graph, which requires frequent distance evaluations between pairs of points. Solving these 2PBVP problems is expensive, and a natural idea is to learn models to reduce the computational cost. One approach is to learn a reachability function. In [2] a large number of 2PBVP problems are solved offline and used to train functions for estimating the cost-limited reachable set. Experiments using a binary SVM and locally weighted linear regression (LWR) yield a 4 order-of-magnitude speedup over exact methods with less than a 10% error rate.

Going one step further, one can learn a distance function. In [5] an obstacle-free distance function is approximated by locally-weighted projection regression (LWPR) using high-quality training samples generated with iterative linear quadratic regression (iLQR). The RRT algorithm is modified to use the learned distance function to select the nearest node for expansion, then iLQR is used to connect to the target state. Experiments for a simple pendulum swing-up problem show the number of node expansions can be significantly reduced.

In [44] a obstacle-free distance function is learned for a 2-wheeled differential drive robot using training samples from a high-quality POSQ solver. Using hand-engineered features with a neural network regression estimator results in a 5 order-of-magnitude decrease in planning time with negligible effect on path quality.

Another approach is to learn to generate control trajectories. In [16] a set of motion primitives in the workspace is learned from an expert tutor for a simple kitchen manipulation task, using fitted cubic polynomials. Then candidate paths for new problems are generated by finding applicable motion primitives. Using this approach yielded both a higher success rate at finding solutions and a lower number of nodes expanded.

RRT-CoLearn is described in [62] which learns indirect optimal controls using costates. This results in a much smaller set of parameters to describe the input control function, in contrast to direct optimal control that requires a series of parameter inputs over time. The paper also proposes a method of filtering the training samples to address a fundamental problem of learning control sequences, which is that the average of two control sequences that yield similar results may yield entirely different results. For example one sequence may steers left around an obstacle while another steers right, but the average of the two trajectories leads straight into the obstacle.

Most of these supervised learning techniques require a high-quality hand-engineered 2PBVP solver to generate the training samples, as well as a large amount of offline time to generate the samples and train the models.

2.5 Obstacles

Many learning approaches utilize supervised training of a model using an exhaustive set of optimally-solved problem samples in an obstacle-free environment. Yet the particular layout of obstacles in a space has an enormous impact on how planning behaves. In regions with low obstacle density and ample space to navigate between obstacles, effective global guidance can be achieved using these obstacle-free distance estimates.

However, as obstacle density increases, one encounters more instances of dead-ends or local minima in which distance heuristics become trapped. Mazes represent an extreme example of spaces with frequent and deep dead-ends. In these cases, relying on obstacle-free distance estimates to distant targets is of little use since the true distance function is much more complex. A better option may be to focus on rapid local exploration instead.

Hence, a planner presented with problems drawn from environments with few obstacles naturally should focus on different strategies than the same planner when presented with problems from environments with dense obstacles or obstacles with regular structure.

Yet no existing work appears to adapt in response to the distribution of obstacles in the environment.

2.6 Global Guidance

Standard sampling-based planning methods consist of some combination of local steering toward a goal, or randomly directed exploration. No global guidance is provided to the algorithm.

One approach to providing global guidance is to analyze the workspace to derive a global distance heuristic. The winning team in the 2012 Physical Traveling Salesman Problem (PTSP) competition used Monte-Carlo Tree Search (MCTS) guided by a global distance heuristic [46]. The heuristic used a scanline floodfill algorithm on the 2D representation of the obstacle map to determine the minimum distance to each goal from any location on the map.

Another approach is to first form a discrete abstract graph representing the search space, then use standard graph search to find an abstract solution path. The abstract solution is then used to guide a local steering controller to find a concrete solution to the problem. An example is [19] which considers a physics-based simulation of a wheeled vehicle navigating in a complex unstructured environment with obstacles with the objective of visiting multiple goal locations. The workspace is first decomposed into an abstract graph, then a modified Traveling Salesman Problem (TSP) solver is used to search for dynamically feasible solutions guided by the abstract graph topology.

The examples cited here make use of fast geometric algorithms for completely partitioning or traversing a discretized 2D space. Unfortunately for problems with 3 or more dimensions, the computational complexity of geometric partitioning grows exponentially and fast exact solutions are generally impossible [37].

Given the high-dimensional nature of kinodynamic planning problems, it seems natural to explore sampling-based methods as a basis for forming global abstractions that might help guide a search that is otherwise randomly exploring. Yet it does not appear any existing work has done so.

2.7 Planning Architectures

A number of previous planners have utilized the concept of abstract states and hierarchies.

The *Skills, Tactics and Play* (STP) architecture [7] provides a hierarchical framework in which domain-specific skills can be composed to solve complex problems quickly, in particular Robot Soccer. In STP, *skills* are specialized low-level control policies designed to achieve specific short-term objectives. *Tactics* utilize multiple skills in a finite state machine to compose higher-level control policies for an agent. *Plays* are team plans for multiple agents and assign coordinated tactics to agents. Planning in this architecture is primarily reactive and designed for speed. Deliberative planning is limited to selecting one play at a time from the playbook without looking further into the future, and hence cannot pursue multiple goals.

[67] integrates RRT methods with STP in order to better exploit control policies for exploration. The approach can solve challenging physics-based problems with moving and adversarial obstacles, but does not consider the problem of reaching multiple goals.

The SyCLoP (synergistic combination of layers of planning) architecture [49] speeds up continuous sampling-based exploration by utilizing a discrete workspace decomposition. Exploration is biased toward extending the motion tree from a selected region to neighboring regions. Repeated failure to connect regions increases the estimated cost and leads the planner to explore alternate routes.

A combination of sampling-based motion planning with efficient TSP solvers has been applied to multi-goal planning with a physics-based game engine in [19]. The problem domains studied consist of vehicles navigating in a cluttered environment to reach multiple goals. The workspace is first decomposed into a triangular mesh of regions, then heuristic estimates are computed based on partial TSP tours using an undirected graph of the mesh geometry. A sampling-based algorithm grows a motion tree starting from the initial state, and heuristic estimates are used to bias growth of the tree along potential TSP tours.

Similar approaches have been applied for the Physical Traveling Salesman Problem (PTSP) [47], a 2-D real-time game where the agent must steer a ship around obstacles to visit as many waypoints as possible within a time limit. The multi-objective version seeks to minimize time, fuel consumption, and damage incurred from collisions. This game featured in three international IEEE competitions in 2012 and 2013. The winning entry in 2013 [50] utilized a global TSP tour based on a geometric analysis of the workspace. Local steering to follow the TSP tour was implemented using MCTS.

2.8 Online Learning and Local Neighborhood Bias

Sampling-based planning offers a number of opportunities to incrementally accumulate knowledge during planning that may be useful for function approximation. For example, the results of collision checking each random sample can be used to build an online approximate model of free configuration space and improve performance by reducing the need for more expensive collision checking [8]. Similarly, when attempting to extend nodes in a search tree, the success or failure may be recorded and over time may be useful for estimating the reachability of regions of space.

Non-parametric function estimation methods represent functions using a set of training samples and are ideally suited for incremental online learning of complex functions with limited training data.

Among these methods, k-Nearest Neighbor (kNN) algorithms have long been used for regression and classification. They are simple, fast and often perform as well or better than more complex methods [21, 64].

kNN methods estimate a function based on the k-nearest neighbors of a query point in a set of training samples. Most of the time Cartesian distance over all or a selected subset of the data features is used to identify the nearest neighbors but other distance metrics have also been used, including learned ones [61, 60]. Typically the estimate is a simple average of the neighbor values, or an inversely-weighted average based on distance from the query point. More generally, the estimator may use locally-weighted regression to fit a function to the neighbors.

The choice of distance metric is crucial for nearest-neighbor algorithms and much effort has focused on learning distance metrics for classification tasks. Representative global methods are described in [66] and [61] that apply global linear transformations to the input data. Locally discriminative transformations are applied in [26] and [41]. In [59] the distance to each training point is scaled inversely with its distance from the class boundary, resulting in fewer neighbors from high-variance regions.

Other approaches vary the size of the neighborhood. In [22] locally-adaptive k values are determined for each training sample. In [45] the local k-neighborhood is extended to include training samples whose k-neighborhood includes the query point, thus increasing the influence of neighbors from lower-density regions.

Standard kNN approaches only consider the distance of the k-nearest neighbors, not their spatial distribution, and implicitly assume a uniform distribution around the query point. Yet local variation in sample density is common, arising naturally when training samples are randomly distributed in the input space. Variation may also result from bias in the training data collection process. For example, when learning a function of a dynamic physical system, samples from regions near unusual states of the system may be sparse relative to the normal operating range. Likewise, regions near the boundary of a training domain are less dense than the interior. As a result, kNN estimates are more heavily influenced by regions of higher sample density within the local neighborhood, since those regions are over-represented in the set of k-nearest neighbors. In general, this *local neighborhood bias* is not desirable for locally smooth regression functions since there is no a priori reason to prefer samples in any particular region.

Yet, no existing methods directly take into account asymmetric distribution of samples in the local neighborhood.

CHAPTER 3

LEARNING HEURISTIC DYNAMICS

3.1 Introduction

Heuristic state-space search for classical planning has demonstrated state-of-the-art performance since the introduction of HSP at AIPS98 [6]. At the 2014 International Planning Competition, the majority of planners utilized heuristic search directly or indirectly.

It is well-known that the performance of a heuristic varies significantly across different types of problems. In fact, the No Free Lunch theorem shows that no one heuristic dominates any other when averaged over all possible search problems [63]. In the context of planning we observe that heuristic performance varies significantly by planning domain and by individual problem. Even within a problem, the informativeness of a heuristic may vary across regions of the search space.

A modern heuristic search planner may utilize multiple heuristics, heuristic parameters, search strategies, and exploration strategies. Furthermore, a portfolio planner may utilize multiple base planners to solve a given problem. In such a context, the question naturally arises of how a planner can identify the best available heuristic, search strategy or base planner for a given situation.

To answer this question we propose a new approach using the information provided by *search dynamics* during planning. By observing search progress over a window of time, we can extract useful features for learning and online decision-making during planning.

3.2 Search Dynamics

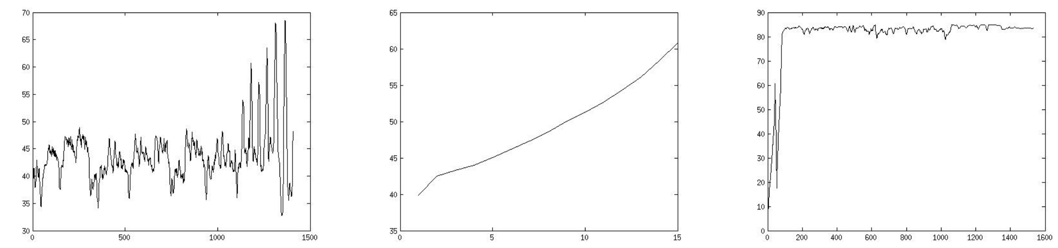
A state-space heuristic search algorithm inherently traverses a sequence of nodes corresponding to states. By treating this sequence of nodes as a signal, one can infer information about the progress of the search. For example, we can observe the sequence of heuristic values (h-values) over time to determine if a heuristic search is stuck in a plateau or local minima.

The search dynamics of a planner depend not only on the heuristic but also on the search algorithm. For example the behavior of greedy best-first search (GBFS) may be quite different than enforced hill-climbing (EHC) using the same heuristic. Furthermore, planners may alternate between local and global search, and may incorporate a variety of random exploration strategies. See for example [65]. All of these affect the search dynamics.

In traditional planners, heuristic progress is typically evaluated based on how many nodes have been expanded since the most recent minimum h-value was achieved. While this is simple and fast, it neglects activity between minima as well as other search node properties. As a result it might not correctly capture the actual progress, especially in problems with weak or inconsistent heuristics.

Figure 3.1 illustrates the dynamics of the g-values for 3 different heuristic planners on a problem from the IPC scanalyzer-sat11-strips domain. The g-values in this domain correspond to the search depth. Visualizing the planner dynamics over time clearly shows information that is not captured by a single node measurement, and thus might be useful to make better progress predictions.

As a result we are motivated to observe the dynamics of sequential node expansions over time. For a given heuristic searching in a specific domain, we ask whether search dynamics can reveal patterns of activity that distinguish when the heuristic planner is making progress or is stalled.



(a) h_{cg} GBFS (b) h_{FF} GBFS (c) LAMA 2011

Figure 3.1: g-value vs. nodes expanded for scanalyzer-sat11-strips p16.pddl

In this work, we present a dynamic portfolio planner that selects from a set of base heuristic search planners during the planning process. Online selection is performed using a learned function to evaluate which heuristic planners are making progress based on the current search dynamics.

3.3 Approach

In contrast to existing approaches, we evaluate the dynamics of each heuristic planner over fixed search windows. During planning, we use these dynamics with a learned regression function to predict after each search window whether the heuristic planner is likely to solve the problem.

We present here the DH1 dynamic portfolio planner. The planner is given a fixed time limit for each problem with the goal of maximizing coverage, i.e. solving as many problems as possible. The planner does not consider plan quality, but it is straightforward to extend the algorithm presented here with base planners that utilize anytime planning techniques such as iterative weighted-A* .

The planner is provided with a set of base heuristic planners H and a representative set of training problems T.

Using the training problems, we learn a regression function for each base planner that estimates distance to solution based on the current heuristic search dynamics. Once trained, the regression function is used during planning to dynamically select base planners that appear to be making the most progress toward a solution.

3.3.1 Feature Extraction

Each base planner $h \in H$ is modified to record data during planning. For each state node expanded during the search phase, a function $\nu : \textit{node} \mapsto \mathbb{R}^l$ captures a vector of l node features.

In order to capture dynamics spanning multiple search nodes, sequences of nodes are further processed in windows of length m. A function $\omega : \mathbb{R}^{l \times m} \mapsto \mathbb{R}^n$ extracts a set of n search dynamics features from each window. These features are subsequently used to predict planner progress.

3.3.2 Training

Algorithm 4 is used to train a regression function $\rho(h) : \mathbb{R}^n \mapsto \mathbb{R}$ for each base planner h. The objective of $\rho(h)$ is to predict distance to the solution based on the current search dynamics, with 0 representing very far from a solution, and 1 very close.

To generate the training set for learning the regression function, each base heuristic planner $h \in H$ is run on a set of training problems T. For each planning problem $\Pi \in T$, a sequence of window feature vectors Φ is generated during the search. Algorithm 6 shows the processing performed for each window. For each $\varphi \in \Phi$ both the feature vector φ and the current search time are recorded. After the training problem is either solved or times out, a set of samples Σ is updated with one sample per feature vector φ. A regression value σ is associated with each sample,

Algorithm 4 DH1 training algorithm

```
input: a set of training problems $T$
  a set of base heuristic planners $H$
  node feature extraction function $\nu : node \mapsto \mathbb{R}^l$
  window feature extraction function $\omega : \mathbb{R}^{l \times m} \mapsto \mathbb{R}^n$
output: $\forall h \in H$:
      prior probability $\pi(h)$
      regression function $\rho(h) : \mathbb{R}^n \mapsto [0, 1]$
for $h \in H$ do
    $\Sigma \leftarrow \varnothing$
    for $\Pi \in T$ do
        $initialize(\Pi, h)$
        $\Phi \leftarrow \varnothing$
        repeat
            $solved, \varphi \leftarrow executePlannerWindow(h)$
            $\Phi \leftarrow \Phi \cup (\varphi, timeElapsed)$
        until $solved$ or $timeElapsed > timeLimit$
        for $\phi \in \Phi$ do
            if $solved$ then
                $\Delta_{time} \leftarrow timeElapsed - \phi.timeElapsed$
            else
                $\Delta_{time} \leftarrow 2 * timeLimit$
            $\sigma \leftarrow 1/(1 + e^{-k*(\Delta_{time} - timeLimit/2)})$
            $\Sigma \leftarrow \Sigma \cup (\varphi, 1 - \sigma)$
    $\pi(h) \leftarrow |\Pi : \Pi \in T, solved(\Pi)| \,/\, |T|$
    $\rho(h) \leftarrow trainRegressionFunction(\Sigma)$
return $\pi, \rho$
```

computed based on the time-to-solution Δ_{time} for each sample. For solved problems, Δ_{time} is simply the difference between the total solution time and sample time. For problems that time out, Δ_{time} is set to twice the time limit. From this, we compute a normalized regression value σ from Δ_{time} using a sigmoidal function to ensure σ ranges (0..1).

After samples Σ have been generated using h for all training problems, the regression function $\rho(h)$ is trained using Σ and an appropriate regression algorithm.

3.3.3 Planning

The dynamic heuristic planner basic algorithm (DH1) is shown in Algorithm 5. DH1 uses the learned regression functions to distribute processing time between the base heuristic planners.

For every $h \in H$, DH1 maintains a preference value ranging [0..1]. A value of 0 indicates the heuristic appears very unlikely to solve the problem, and 1 indicates the heuristic is very likely to solve the problem. We initialize each $preference(h)$ based on the prior probability $\pi(h)$ of planner h solving a problem in the domain as determined during training.

After initialization, the search process proceeds one feature window at a time. At each step the planner h to use is selected based on the $preference(h)$ values. The selected planner executes one search window using Algorithm 6 and the window feature vector φ is computed. Based on this, the pre-trained regression function $\rho(h)$ estimates the distance-to-solution δ. Finally $preference(h)$ is updated with δ.

This means that in effect, the base planners are run in parallel and the DH1 planner switches between them and allocates processing time based on the estimated search progress. No information is exchanged between the base planners.

3.4 Experiments

Experiments were conducted on 8 domains taken from IPC competitions. For each domain, the DH1 planner was trained using a randomly selected 75/25 training/test split. Tests were run on Intel Dual-Core I3-500 3.2 GHz processors with a memory limit of 4GB and a timeout of 60 seconds per problem. All results are averaged over 10 independent experiments.

Algorithm 5 DH1 planning algorithm

```
input: planning problem $\Pi$
  a set of base heuristic planners $H$
  node feature extraction function $\nu : node \mapsto \mathbb{R}^l$
  window feature extraction function $\omega : \mathbb{R}^{l \times m} \mapsto \mathbb{R}^n$
  $\forall h \in H$:
      prior probability $\pi(h)$
      regression function $\rho(h) : \mathbb{R}^n \mapsto [0, 1]$
output: a plan, or $\perp$
$initialize(\Pi, h)$
for $h \in H$ do
    $preference(h) \leftarrow \pi(h)$
while not $timeout$ do
    $h \leftarrow selectPlanner(H, preference[])$
    $solved, \varphi \leftarrow executePlannerWindow(h)$
    if $solved$ then
        return $h.plan$
    $\delta \leftarrow \rho(h, \varphi)$
    $preference(h) \leftarrow updatePref(preference(h), \delta)$
return $\perp$
```

Algorithm 6 executePlannerWindow

```
input: planning problem $\Pi$
  base heuristic planner $h$
  node feature extraction function $\nu : node \mapsto \mathbb{R}^l$
  window feature extraction function $\omega : \mathbb{R}^{l \times m} \mapsto \mathbb{R}^n$
output: $true$ if a plan is found, else $(false, \varphi)$ where $\varphi$ is the window feature vector
for $i : 1, m$ do
    $node \leftarrow fetchNext(h.openList)$
    if $isGoal(\Pi, node)$ then
        return $true$
    $window[i] \leftarrow \nu(node)$
    $updateOpenList(h, node)$
$\varphi \leftarrow \omega(window)$
return $(false, \varphi)$
```

Domain selection was performed on the following basis. The base planners were first tested independently on all IPC domains. For each domain, if a base planner solved every problem solved by any of the rest of the base planners, we say that planner *dominated* the domain. Domains with one or more dominant base planners were then excluded as uninteresting. Since the planner is working with a fixed set of base heuristic planners, one cannot improve on the policy of simply picking a dominant base planner for all problems in the domain. Instead domains were selected in which it is more difficult to predict which base planner will solve a particular problem.

3.4.1 Heuristics

The following base heuristic planners were configured using the Fast Downward planner [27]:

- FF/additive heuristic h_{FF} with lazy GBFS [30]
- Context-enhanced additive heuristic h_{cea} with lazy GBFS [28]
- Causal graph heuristic h_{cg} with lazy GBFS [27]
- Alternating h_{FF} and h_{cea} with lazy GBFS
- LAMA 2011 [53]
- An experimental modification of h_{FF} that counts violated preconditions in the relaxed plan, with lazy GBFS

These were selected based on the diversity of the underlying heuristics and their good performance but without an attempt of being comprehensive or optimizing diversity or ensemble performance.

3.4.2 Features

The base planners were modified to record data during planning. The node feature extraction function ν recorded the following features for each search node encountered:

- h-value (cost-to-go)
- g-value (cost-to-reach)
- Whether the node is a dead-end

All experiments used a window size of 200. The pick of this window size was informed by some knowledge of solution length and planner run times but no empirical study of different sizes or window size optimization was used. The window feature extraction function ω generated the following features:

- Mean h- and g-values
- Minimum h- and g-values
- Maximum h- and g-values
- Standard deviation of h- and g-values
- Slope of h and g
- Number of dead-ends
- Number of nodes expanded since most recent minimum h-value achieved
- Log of the total nodes expanded
- Time elapsed during the window

These initial features can easily be extended but we considered them sufficient for experimentation. No feature selection or optimization process was performed.

3.4.3 Training and Parameters

The domain-specific regression functions $\rho(h)$ were learned using ϵ-SVR support vector regression using a Gaussian kernel with parameters determined by grid-search.

Preference values $preference(h)$ were updated with distance estimates δ using exponentially-weighted averaging with a weight-decay factor λ to reduce noise:

$$preference(h) \leftarrow (1-\lambda) * preference(h) + \lambda * \delta$$

λ was set to 0.7 based on empirical testing using a range of values.

After each planning window, the next planner h to use is stochastically selected using a Boltzmann soft-max function applied to the $preference(h)$ values. The probability $P(h)$ of selecting planner h is calculated by

$$P(h) = \frac{e^{\frac{preference(h)}{\tau}}}{\sum_{h' \in H} e^{\frac{preference(h')}{\tau}}}$$

The temperature τ was set to 0.1 based on empirical testing using a range of values.

3.4.4 Results

To demonstrate the potential of the DH1 planner and to evaluate its performance, it was compared to 3 fixed planner selection strategies, namely Best Prior, Equal and Ideal.

Best Prior simply selects the base planner that had the best performance on the training set for the domain. Equal divides the available search time equally between all of the base planners. Ideal selects one of the base planners that can solve the problem, if any. Note that Ideal represents the best possible planner selection and would require an oracle. It is not a practical strategy but an upper bound on performance.

The results shown in Figure 3.2 represent the average performance over 10 experiments, each with an randomly partitioned 75/25 training/test split. For each

Domain	**Best Prior**	**Equal**	**DH1**	**Ideal**	**Total**	**NonDom**
airport	8.60	8.20	8.18	8.90	12	1
depot	4.00	3.90	4.00	4.50	5	1
parcprinter-sat11-strips	3.10	5.00	5.00	5.00	5	6
pathways	5.50	6.30	6.30	6.30	7	3
pipesworld-tankage	7.90	9.20	9.43	10.30	12	6
scanalyzer-sat11-strips	4.50	4.70	4.20	5.00	5	1
tidybot-sat11-strips	2.40	2.30	2.70	3.30	5	2
transport-sat11-strips	2.60	2.00	3.17	4.00	5	3
Total	38.60	41.60	42.98	47.30	56	23

Figure 3.2: Experimental Results

experiment, each selection strategy was applied to the test problems in the domain and the number of problems solved within the 60 second time limit were recorded.

The average number of test problems solved by each of the strategies is shown in their respective columns. The Total column is the total number of test problems per experiment. The NonDom column is the total number of non-dominated problems in the domain, i.e. the number of problems that were not solved by the Best Prior planner but were solved by at least one other base planner.

The overall results show DH1 performs better than both the Best Prior and Equal strategies.

DH1 achieves Ideal performance in the parcprinter and pathways domains, but underperforms Best Prior and Equal in the airport and scanalyzer domains. The airport and scanalyzer domains each contain only 1 non-dominated problem and as a result Best Prior is difficult to beat as it is close to Ideal. DH1 performs relatively better in domains with more non-dominated problems. In these domains there is more variability and it is more difficult to predict *a priori* which base heuristic planner will solve a given problem.

3.5 Conclusions and Further Work

By observing heuristic search progress over a period of time, we can extract features of the search dynamics that are useful for learning and online decision-making during planning.

This chapter presents DH1, a novel algorithm that learns from the search dynamics for a set of base heuristic planners. This knowledge is used during planning to dynamically select from the base planners. Results are promising and show that performance can be improved over static approaches that select base planners before planning starts.

Future work may explore further properties and applications of heuristic search dynamics in planning. DH1 is a portfolio planner and the base heuristic planners are entirely independent of one another. Search dynamics are used only for switching between base planners. Still to be investigated is the use of search dynamics within a base planner to select between multiple heuristics with shared open lists instead of using fixed strategies such as alternation. This may improve performance if some heuristics are more informative than others depending on the region of the search space.

In DH1 the regression function is used solely for selecting a base heuristic planner to execute. Another question is whether the regression estimator can be used to better determine when a planner should utilize random exploration or a deep local search in order to escape a local minima in conjunction with algorithms such as [65].

Lastly, the window feature extraction function $\omega : \mathbb{R}^{l \times m} \mapsto \mathbb{R}^n$ used here is hand-coded and fixed. An open question is whether more effective features can be learned directly from the data.

CHAPTER 4

BALANCED NEAREST NEIGHBORS

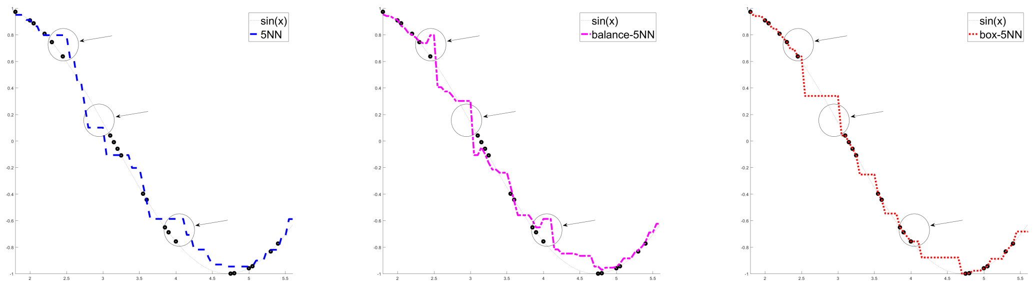
4.1 Introduction

To address the problem of local neighborhood bias described in Section 2.8, this work introduces two new kNN algorithms, Axis-balanced kNN and Box kNN, that use sample balancing, and investigates the effect of different sample attributes on their performance relative to standard kNN.

Figure 4.1a illustrates a 1-dimensional example. A random distribution of noise-free sample points are taken from the function $y = sin(x)$. In the highlighted regions, 5-NN overestimates or underestimates the function because most of the neighbors are on one side of the query point. In other regions where the nearest neighbors are approximately equally distributed in either direction, the estimate is much closer.

Motivated by this observation, we can adjust the neighbor weights to approximate an equal distribution along each feature axis in the input space. In this example, if 4 neighbors are in the negative x direction, and 1 neighbor is in the positive x direction, we can adjust the relative weight of the positive neighbor by a factor of 4. Intuitively, that neighbor is likely to provide more information since it is in a region that is under-represented among the neighborhood samples. Figure 4.1b shows results of the Axis-balanced 5-NN algorithm described below using the same sample points as 4.1a. The mean-squared error (MSE) is reduced by 11% in this example.

A further observation in this example is that if we have noise-free samples and multiple neighbors are in the same direction from the query point, the nearest neighbor in that direction will be the most informative. Additional neighbors further away



(a) 5-NN (b) Axis-balanced 5-NN (c) Box 5-NN

Figure 4.1: Local Neighborhood Bias

in the same direction may reduce accuracy rather than improve it. This motivates us to consider at most one neighbor in the direction of each axis from the set of nearest neighbors, using the subset of neighbors that form an axis-aligned bounding box around the point. Figure 4.1c shows the effect of this Box 5-NN algorithm. The MSE in this case is reduced by 58%.

4.2 Approach

For real-valued outputs $y \in \mathbb{R}$, regression seeks to approximate a function $f : X \mapsto \mathbb{R}$ where $X \in \mathbb{R}^d$ is a metric space with a distance function $\delta : X \times X \mapsto \mathbb{R}$. Similarly, for categorical outputs $y \in C$, classification seeks to approximate a function $f : X \mapsto C$.

Let S be a set of N training samples $\{x^{(i)}, y^{(i)}\}_{i=1}^{N}$ where $x^{(i)} \in X$ is an input variable and $y^{(i)}$ is the corresponding output variable. For a query point $q \in X$ and distance function δ, define the ordered set $A \subset S$ of *k-nearest neighbors* $\{a^{(i)}\}_{i=1}^{k}$ such that:

$$|A| = k$$

$$\forall a \in A, b \in S - A, \delta(a.x, q) \leq \delta(b.x, q)$$

$$\forall a^{(i)}, a^{(j)} \in A, i < j, \delta(a.x^{(i)}, q) \leq \delta(a.x^{(j)}, q)$$

Let $w^{(i)}$ be the weight associated with neighbor $a^{(i)}$.

Let ω be an initial weighting function that calculates weights $\omega^{(i)}$ from A, $\omega : \mathbb{R}^d \times \mathbb{R}^{d\times k} \mapsto \mathbb{R}^k$.

The standard formulation for kNN regression uses a weighted average of the k-nearest neighbors from a set of training samples. For query point $q \in X$, the true function $f(q)$ is estimated by:

$$\hat{f}(q) = \frac{1}{Z_q} \sum_{i=1}^{k} w^{(i)} y^{(i)} \tag{4.1}$$

where $y^{(i)}$ is the output value of the i-th nearest neighbor, $w^{(i)}$ is the corresponding weight for each point resulting from the used weighting function, and $Z_q = \sum_{i=1}^{k} w^{(i)}$ is a normalization factor.

Similarly, kNN classification predicts discrete class labels $y \in C$ as:

$$\hat{f}(q) = \underset{c \in C}{\text{argmax}} \sum_{i=1}^{k} \begin{cases} w^{(i)} & \text{if } y^{(i)} = c \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Basic kNN weights all k-nearest neighbors equally with $w^{(i)} = 1/k$. Effectively it assumes a uniform function value in the local neighborhood spanned by the k neighbors.

Distance-weighted kNN (w-kNN) incorporates the idea that neighbors closer to the query point should have a greater influence than neighbors farther away. A wide variety of weighting functions based on distance have been investigated [18, 58]. A simple choice is to use the inverse of the distance, $w^{(i)} = 1/\delta(x^{(i)}, q)$.

4.2.1 Axis-balanced kNN

The Axis-balanced kNN algorithm uses Equations (4.1) and (4.2) but modifies the weighting factors $w^{(i)}$ as follows.

For query point q, find the k-nearest neighbors A. For each $a^{(i)} \in A$, assign weight $w^{(i)}$ using some initial standard weighting function ω. Now for each dimension $j \in [1..d]$, partition the neighbors in each direction along axis j.

$$\begin{aligned} L_j &= \{a^{(i)} : a^{(i)} \in A, a.x_j^{(i)} < q_j\} \\ R_j &= \{a^{(i)} : a^{(i)} \in A, a.x_j^{(i)} > q_j\} \\ E_j &= \{a^{(i)} : a^{(i)} \in A, a.x_j^{(i)} = q_j\} \end{aligned} \tag{4.3}$$

If $|L_j| > 0$ and $|R_j| > 0$ then adjust weights $w^{(i)}$ using:

$$w^{(i)} \leftarrow \begin{cases} w^{(i)}(|L_j| + |R_j|)/|L_j| & \text{if } a^{(i)} \in L_j \\ w^{(i)}(|L_j| + |R_j|)/|R_j| & \text{if } a^{(i)} \in R_j \\ w^{(i)} & \text{if } a^{(i)} \in E_j \end{cases} \tag{4.4}$$

Compared to standard kNN the additional computation time is $O(dk)$.

4.2.2 Box kNN

The Box kNN algorithm also uses Equations (4.1) and (4.2) but modifies the weighting factors $w^{(i)}$ as follows.

For query point q, find the k-nearest neighbors A. For each $a^{(i)} \in A$, assign weight $w^{(i)}$ using some initial standard weighting function ω. Now for each dimension

Algorithm 7 Axis-balanced kNN regression

```
input: set of training samples $S \in \mathbb{R}^d \times \mathbb{R}$
 query point $q \in \mathbb{R}^d$
 distance function $\delta : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$
 weighting function $\omega : \mathbb{R}^d \times \mathbb{R}^{d \times k} \mapsto \mathbb{R}^k$
output: Predicted $y \in \mathbb{R}$

$A \leftarrow findNeighbors(q, S, k, \delta)$
$W \leftarrow \omega(q, A)$
for $j \in [1..d]$ do
    $l \leftarrow 0, r \leftarrow 0$
    for $i \in [1..k]$ do
        if $A[i].x_j < q_j$ then $l \leftarrow l + 1$
        else if $A[i].x_j > q_j$ then $r \leftarrow r + 1$
    if $l > 0$ & $r > 0$ then
        for $i \in [1..k]$ do
            if $A[i].x_j < q_j$ then $W[i] \leftarrow W[i](l + r)/l$
            else if $A[i].x_j > q_j$ then $W[i] \leftarrow W[i](l + r)/r$
$y \leftarrow 0$, $z \leftarrow 0$
for $i \in [1..k]$ do
    $y \leftarrow y + W[i]A[i].y$
    $z \leftarrow z + W[i]$
$y \leftarrow y/z$
return $y$
```

$j \in [1..d]$, partition the neighbors in each direction along axis j as in Equation (4.3) and identify nearest neighbors in each dimension as:

$$\tilde{L}_j = \{l \in L_j : l.x_j = \max_{a \in L_j} a.x_j\}$$

$$\tilde{R}_j = \{r \in R_j : r.x_j = \min_{a \in R_j} a.x_j\}$$

Algorithm 8 Box kNN regression

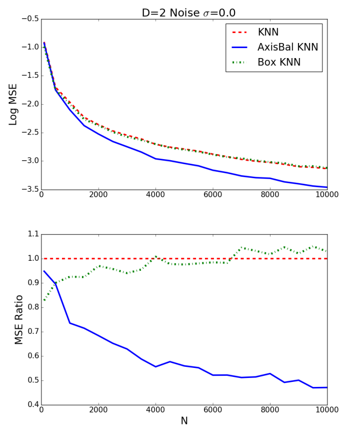
```
input: set of training samples $S \in \mathbb{R}^d \times \mathbb{R}$
 query point $q \in \mathbb{R}^d$
 distance function $\delta : \mathbb{R}^d \times \mathbb{R}^d \mapsto \mathbb{R}$
 weighting function $\omega : \mathbb{R}^d \times \mathbb{R}^{d\times k} \mapsto \mathbb{R}^k$
output: Predicted $y \in \mathbb{R}$

$A \leftarrow findNeighbors(q, S, k, \delta)$
$W \leftarrow \omega(q, A)$
for $j \in [1..d]$ do
    $l \leftarrow -\infty, r \leftarrow \infty, L[j] \leftarrow \varnothing, R[j] \leftarrow \varnothing$
    for $i \in [1..k]$ do
        if $A[i].x_j \leq q_j$ then
            if $A[i].x_j > l$ then $l \leftarrow A[i].x_j, L[j] \leftarrow A[i]$
            else if $A[i].x_j = l$ then $L[j] \leftarrow L[j] \cup A[i]$
        if $A[i].x_j \geq q_j$ then
            if $A[i].x_j < r$ then $r \leftarrow A[i].x_j, R[j] \leftarrow A[i]$
            else if $A[i].x_j = r$ then $R[j] \leftarrow R[j] \cup A[i]$
$y \leftarrow 0$, $z \leftarrow 0$
for $i \in [1..k]$ do
    $c \leftarrow 0$
    for $j \in [1..d]$ do
        if $A[i] \in L[j]$ then $c \leftarrow c + 1$
        if $A[i] \in R[j]$ then $c \leftarrow c + 1$
    $y \leftarrow y + cW[i]A[i].y$
    $z \leftarrow z + cW[i]$
$y \leftarrow y/z$
return $y$
```

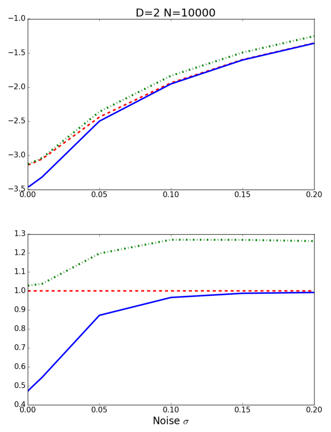
Using this, adjust weights $w^{(i)}$ using:

$$w^{(i)} \leftarrow w^{(i)} \sum_{j=1}^{d} \begin{cases} 2 & \text{if } a^{(i)} \in E_j \\ 1 & \text{if } a^{(i)} \in \tilde{L}_j \cup \tilde{R}_j \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

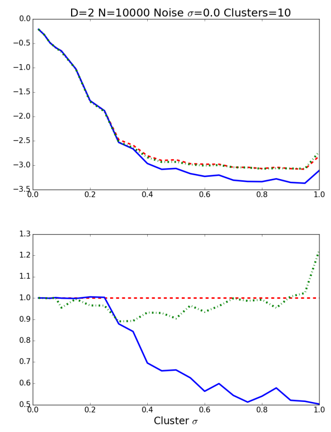
As with Axis-balanced kNN, the additional computation time is $O(dk)$.



(a) Effect of Sample Density



(b) Effect of Noise



(c) Effect of Cluster Density

Figure 4.2: Synthetic Datasets: Top Row is Log MSE, Bottom is Ratio of MSE vs. kNN

4.3 Synthetic Experiments

To study the effects of different dataset properties on the estimation accuracy of the kNN algorithms, synthetic datasets were generated using the radial sin function $f(x) = sin(2\pi\|x\|)$ for sample points generated in a unit hypercube. The standard and balanced kNN algorithms were all distance-weighted using inverse Euclidean distance as the initial weighting function $\omega^{(i)} = 1/\|q, x^{(i)}\|$.

Each experiment performed 5-fold cross-validation to measure performance and was repeated until reaching confidence level $P > 0.99$ that the relative error is less than 0.01. Results are analyzed in terms of Mean Square Error (MSE) and are also presented as the ratio of MSEs compared to standard kNN to make comparisons easier.

4.3.1 Effect of Uniform Sample Density

Figure 4.2a compares the algorithms for a 2-dimensional space, over a range of uniform sample densities. In areas with very low density and low noise, Box kNN works best. As density increases, Axis-balanced kNN performs increasingly better.

4.3.2 Effect of Noise

Figure 4.2b compares the algorithms for a 2-dimensional space, over a range of noise levels. Box kNN performs relatively poorly for noisy data. This is not surprising since it considers only a subset of its nearest neighbors, resulting in higher variance in its estimates. Axis-balanced kNN performance is more tolerant of noise, but also deteriorates relative to kNN as noise increases.

4.3.3 Effect of Non-uniform Sample Density

In order to observe the effect of a non-uniformly generated training set, training sample points were drawn from 10 normally distributed clusters $x \sim N(\mu_i, \sigma_{cluster})$ and the density of the clusters was varied by varying the standard deviation $\sigma_{cluster}$. The test sample points were drawn from a quasirandom uniform distribution over the unit hypercube using a Hammersley sequence.

Figure 4.2c compares the algorithms for a 2-dimensional space, over a range of $\sigma_{cluster}$ densities.

For small dense clusters, Box kNN works best. As cluster density decreases and the training distribution becomes more uniform, Axis-balanced kNN performs increasingly better. It should be noted that one of the balanced kNN algorithms always outperformed standard kNN on these synthetic datasets.

4.4 Real-World Experiments

While synthetic datasets allow to study the effects of different data sample distributions on the accuracy in the context of a smooth function, they do not capture the effects that diversity in real-world function and task have on performance. To address this, experiments were performed on 11 regression datasets and 3 classification datasets obtained from the UCI machine learning repository [15].

Each experiment performed 5-fold cross-validation to measure performance and was repeated until reaching confidence level $P > 0.99$ that relative error is less than 0.01.

kNN methods using isotropic distance functions perform relatively poorly when given uninformative or redundant input features. Consequently some form of feature selection and scaling is often required to obtain good accuracy. Here, simple forward and backward stepwise feature selection was performed to obtain the best features for each combination of dataset and algorithm [21]. In addition, each experiment was performed using the original unscaled features and then with each feature scaled to unit standard deviation, and results are shown using the best scaling option.

Each algorithm was further tested using initial weighting functions $\omega^{(i)} = 1/k$ and $\omega^{(i)} = 1/\delta(x^{(i)}, q)$ and results are shown using the best weighting.

Each experiment varied k from 1 to 40 and selected the value of k with the best accuracy. In order to make a fair comparison, it is necessary to allow different values of k for each algorithm since each approach utilizes the k nearest neighbors differently.

Tables 4.1 and 4.2 show summary results for regression and classification problems. For each dataset the total number of samples and dimensions is shown. Each algorithm shows the number of selected features d and the value of k with the best accuracy.

Dataset	Samples	kNN			Axis-balanced kNN			Box kNN		
		d	K	MSE	d	K	MSE	d	K	MSE
3D Roads	434874	2	2	1.070	2	2	1.059	**2**	**3**	**0.925**
Airfoil Self Noise	1503	5	2	5.10	**4**	**4**	**4.99**	**3**	**30**	**4.95**
Concrete	1030	5	4	48.2	**6**	**4**	**47.3**	5	10	52.4
Energy Cool	768	7	4	3.47	**7**	**4**	**3.20**	4	16	3.88
Energy Heat	768	5	1	0.355	6	4	0.379	**6**	**2**	**0.345**
Parkinson's Motor	5876	**16**	**13**	**44.1**	16	2	56.1	16	13	44.9
Power	9568	4	7	13.5	4	5	15.1	**4**	**18**	**11.8**
Protein	45731	9	6	13.7	9	2	15.1	**9**	**10**	**12.9**
Red Wine Quality	1599	**4**	**31**	**0.326**	4	37	0.346	8	31	0.341
White Wine Quality	4898	**8**	**20**	**0.369**	4	50	0.392	**10**	**31**	**0.369**
Yacht	308	**2**	**8**	**1.95**	**2**	**8**	**1.96**	2	3	2.05

Table 4.1: Real-World Regression Datasets

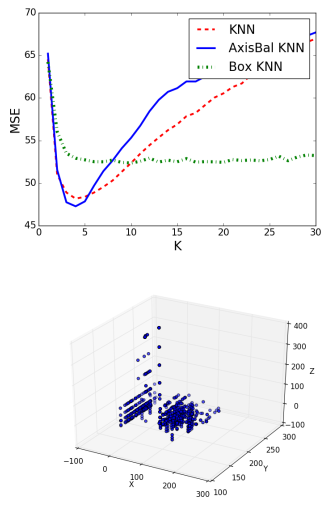
These results show that in the majority of datasets, balancing of the neighbors using one of the two algorithms can improve performance with only two datasets leading to standard kNN having slightly better performance and two sets where it is tied. This illustrates the benefit of balancing but also underlines the importance of selecting the correct scheme for the particular aspects of the dataset.

Visualizing the training samples reveals several datasets consist of samples that are completely or partially organized on a regular grid of feature coordinates. This group includes Airfoil Self Noise, Balance, Concrete, Energy Cooling and Heating, and Yacht.

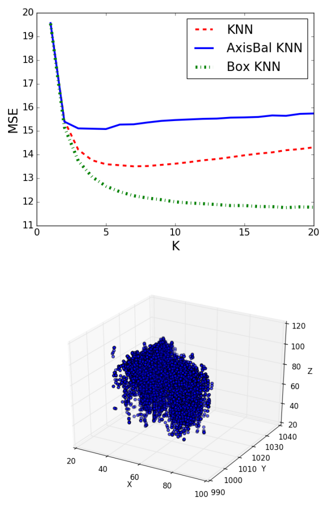
Figure 4.3a shows a representative example from this group. Here the Box kNN algorithm quickly levels off to a constant error since it is effectively restricted

Dataset	Samples	kNN			Axis-balanced kNN			Box kNN		
		d	K	Error %	d	K	Error %	d	K	Error %
Balance	625	4	17	10.06	**4**	**13**	**9.44**	4	11	19.01
Iris	150	4	13	3.15	**4**	**20**	**1.90**	4	1	4.24
Wine	178	**8**	**32**	**0.742**	6	12	3.621	9	11	1.040

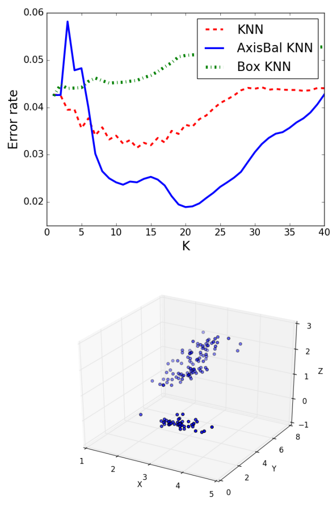
Table 4.2: Real-World Classification Datasets



(a) Concrete



(b) Power



(c) Iris

Figure 4.3: Real-world Datasets. Top Shows Error, Bottom Shows Sample Distribution.

to the neighbors with the same grid coordinates in most dimensions. Axis-balanced kNN and standard kNN achieve highest accuracy with a small k value and rapidly deteriorate as k increases, indicating that the function value changes quickly outside of a small neighborhood.

The remaining datasets consist of samples that are randomly distributed in the feature space. Figure 4.3b shows a representative example. All algorithms exhibit smooth asymptotic behavior and the Box kNN algorithm performs well on these datasets.

Figure 4.3c shows the well-known Iris classification problem in which the samples are clearly clustered, thus leading to a different behavior. The Axis-balanced kNN algorithm performs well here and reduces the error rate by 40% while Box kNN does not perform well, likely due to moderate noise in the data.

Higher values of k suggest increasing noise since they indicate better accuracy is obtained by averaging more samples from a larger neighborhood. The results in

Tables 4.1 and 4.2 show standard kNN is likely the best choice in this case. Balancing in the presence of high noise increases the variance and is less effective.

The results show that the balancing algorithms work best on datasets in which the intrinsic dimensionality is less than around 10 dimensions. Beyond this level the sample density is likely insufficient to perform effective balancing.

In summary this data shows that behavior of the algorithms varies significantly with the data sample distribution and the type of function estimated, thus again reinforcing the importance of the correct choice of balancing scheme for the specific problem and dataset. It also shows the general benefit of balanced kNN approaches when correctly selected, as they can outperform standard kNN for most problems and often show more stability for varying values of k.

4.5 Conclusions and Further Work

This work introduces two modifications of standard kNN. Axis-balanced kNN adjusts the weights of the k-nearest neighbors to approximate a balanced distribution along each feature axis. Box kNN, in contrast, adjusts the weights of the k-nearest neighbors to include only the nearest neighbors in each feature axis direction. Neither method requires additional parameters or tuning beyond that required by kNN.

Experiments using synthetic and real-world data demonstrate these methods can improve accuracy in comparison to kNN. Axis-balanced kNN performs better when training data is more dense and there is little to moderate noise. Box-kNN tends to outperform the other approaches when training data is less dense and there is little noise. When there is a high level of noise, or the intrinsic dimensionality of the data is 10 or higher, standard kNN is likely the best choice.

The balancing here is performed solely along the feature axes of the original sample data. To further improve performance, balancing in other directions may be more effective, such as using whitened data. Another potential improvement is to first globally transform the sample data using distance metric learning methods for classification problems.

CHAPTER 5

ABSTRACTION-GUIDED PLANNING WITH CONTROL POLICIES

5.1 Introduction

Abstraction is a powerful technique for compressing the vast state spaces encountered in continuous planning problems. Solutions found using simplified discrete representations can provide global guidance for lower-level continuous planning components. Yet as described above in Section 2.6, forming useful abstractions of spaces for kinodynamic planning is generally difficult and domain-specific.

Likewise, control in kinodynamic systems is hard due to the vast action space. A controller may be required to specify actions frequently and frame rates of 50Hz or more are not uncommon. A 2-second task running at 100Hz with 3 discrete actions has $3^{200} \approx 2.7 \times 10^{90}$ possible action sequences. For highly-constrained and precise tasks the space of valid solutions is many orders of magnitude smaller. Thus random action sampling alone cannot possibly solve complex problems.

In practice, kinodynamic planners usually rely on domain-specific hand-engineered or learned control policies to select actions. A control policy represents a low-level capability or skill that may be useful in some situations, for example "steer toward a point", "turn a specified angle", "accelerate to a specified velocity", or "jump forward". Composite policies can be formed by concatenating 2 or more existing policies.

This section presents an abstraction-guided planning and execution algorithm that is based on the available control policies. The method requires only a forward simulator of the system dynamics and a predefined set of control policies and no distance metric is required.

Rather than basing abstract states on a simple arbitrary geometric decomposition of space, the approach here forms a useful abstraction of the state space in relation to the control policies. Abstract states correspond to regions in the state space such that any state is reachable from any other state in the region using one of the control policies. Hence, abstract states correspond to regions in the state space that are closed under individual control policies.

The planner explores reachable space using the control policies and builds a directed graph of abstract states. Connectivity between abstract states is established by applying control policies to states sampled from a region and recording transitions to other abstract state regions or goal regions. The resulting graph is used to search for an abstract high-level plan that is dynamically feasible and achieves all of the goals.

The algorithm is implemented and evaluated in a real-time planning agent for the Geometry Friends competition held at the 2019 IEEE Conference on Games. Competition results demonstrate planner performance is comparable to a more highly-engineered domain-specific planner, and significantly outperforms a more general sampling-based planner.

5.2 Formal Problem Statement

Let $\mathcal{S} \subseteq \mathbb{R}^d$ denote the state space and $\mathcal{A} \subseteq \mathbb{R}^k$ denote the action space with system dynamics approximated by a function

$$s' = \textsc{Simulate}(s, a, dt) \tag{5.1}$$

where $s' \in \mathcal{S}$ is the result of applying action $a \in \mathcal{A}$ for a time step of duration dt.

Let $\mathcal{G} = \{G_1, ..., G_n\} \subset \mathcal{S}$ denote a set of n goal regions. Let $\textsc{Goal} : \mathcal{S} \mapsto \mathcal{G} \cup \perp$ specify the goal region, if any, containing a given state.

Let $\Pi = \{\pi_1, ..., \pi_m\}$ be a set of control policies $\pi_i : (\mathcal{S}, \theta) \mapsto \mathcal{A} \cup \perp$, where $\theta \in \mathbb{R}^{l_i}$ denotes parameters for π_i and $\pi_i(s, \theta)$ returns $\perp$ if $\pi_i(\theta)$ terminates in state s.

Starting at state $s \in \mathcal{S}$ and applying a policy $\pi(\theta)$ for $T \in \mathbb{N}$ time steps results in a trajectory $\tau : [1, ..., T] \mapsto \mathcal{S}$, where $\tau(1) = s$ and $\tau(i+1) = \textsc{Simulate}(\tau(i), a_i, dt)$ with $a_i = \pi(\tau(i), \theta)$ for all $i = 1...T-1$. Let $\tau(s, \pi(\theta))$ denote the trajectory starting at state s and applying policy $\pi(\theta)$ until $\pi(\tau(i), \theta) = \perp$.

For any trajectory τ with length T, let $\textsc{First}(\tau) = \tau(1)$ and $\textsc{Last}(\tau) = \tau(T)$. Let $\textsc{Goals}(\tau) = \cup_{i=1}^{T} \textsc{Goal}(\tau(i))$ denote the set of goal regions reached by the trajectory.

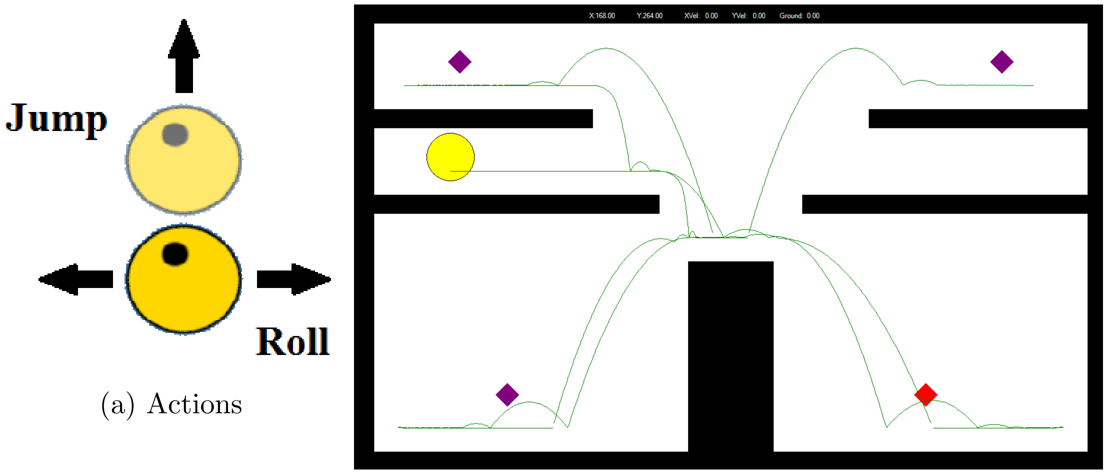
The planning problem can now be defined. Given $\langle \mathcal{S}, \mathcal{A}, \mathcal{G}, \Pi \rangle$ and initial state $s_{init} \in \mathcal{S}$, find actions $[a_1, ..., a_T]$ and trajectory τ such that $\tau(1) = s_{init}$ and $\tau(i+1) = \textsc{Simulate}(\tau(i), a_i, dt)$ for all $i = 1...T-1$, and $\textsc{Goals}(\tau) = \mathcal{G}$.

5.3 Geometry Friends

Geometry Friends (GF) is a challenging physics-based 2-dimensional problem-solving game featured in several recent competitions [51]. The game includes two agents, a Circle and a Rectangle, that may act independently or cooperatively. The implementation here solves problems using only the Circle agent.

The Circle agent is subject to 2-dimensional physics with momentum, friction and gravity. It has sensors that provide it with the current location and linear velocity, but the angular velocity is not directly observable. A set of 4 control actions $\mathcal{A}$ are available at any time step:

- $None$: Apply no force.
- $RollLeft$: Apply a counter-clockwise torque to the Circle.



(a) Actions

(b) Multiple Goals

Figure 5.1: Circle Agent for Geometry Friends

- *RollRight*: Apply a clockwise torque to the Circle.
- *Jump*: Apply an instantaneous upward force to the Circle. This action has no effect if the Circle is not in contact with a surface.

As shown in Figure 5.1, the objective of the game is to collect a set of goals $\mathcal{G}$ represented as purple diamonds as quickly as possible, within a time limit. If all of the goals are collected the level is considered solved. The agent must jump from platform to platform to reach some goals and a high-level plan is required in order to reach all of the goals successfully.

5.3.1 States

The full game state $s \in \mathcal{S}$ consists of $(x, \dot{x}, y, \dot{y}, \omega, O, C)$ where x and y represent the location of the circle character along with their respective velocities, ω is the angular velocity of the circle, O is the obstacle configuration of the level, and C is the set of remaining collectibles (goals not yet achieved).

Sensing provides the agent with $(x, \dot{x}, y, \dot{y}, O, C)$ but ω is not observable. Sensing events during game play are asynchronous and there is a small non-deterministic delay between the time an observation is recorded and the time the agent receives it.

5.3.2 Control Policies

The planner uses a set Π of four low-level hand-engineered control policies that incorporate highly-specific domain knowledge.

5.3.2.1 RollTo

The $RollTo(x_{target}, \dot{x}_{target})$ policy rolls the agent to a target x position and velocity on the current surface.

The policy was learned using value iteration [57]. The action at time step t is selected based on the rolling state $s_t(x_t, \dot{x}_t, x_{target}, \dot{x}_{target})$. A state is a terminal state if $|x - x_{target}| < \epsilon_x$ and $|\dot{x} - \dot{x}_{target}| < \epsilon_{\dot{x}}$ for thresholds ϵ_x and $\epsilon_{\dot{x}}$. Actions are selected from the set of rolling actions $A_r = \{NoAction, RollLeft, RollRight\}$.

Value iteration requires a transition model $P(a, s, s')$ that specifies the probability $\{s_{t+1} = s' | s_t = s, a_t = a\}$. The transition model was learned using a test agent that randomly applied actions from A_r for all possible x velocities and recorded the resulting states.

Value iteration also requires a reward function $R(s)$ that specifies expected reward at each time step when the agent is in state s. The reward function used is:

$$R(s) \leftarrow \begin{cases} 0 & \text{if } s \text{ is a terminal state} \\ -1 & \text{otherwise} \end{cases} \tag{5.2}$$

Using the transition model and reward function, value iteration iteratively updates the value of each state $V(s)$ until convergence. For iteration $k+1$ the update is:

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} P(a, s, s')[R(s) + \gamma V_k(s')] \tag{5.3}$$

where γ specifies a discount factor for future states. The value used for γ is 0.999.

After the state value function $V(s)$ converges, the deterministic control policy $\pi(s)$ is obtained by:

$$\pi(s) = \operatorname*{argmax}_{a} \sum_{s'} P(a, s, s')[R(s) + \gamma V_k(s')] \tag{5.4}$$

The specified reward function with -1 reward per time step is useful because the resulting $|V(s)|$ provides a good estimate of the number of time steps required to reach a terminal state from state s.

The value and policy functions were implemented using a simple tabular representation with discretized states.

5.3.2.2 JumpAndStop

JumpAndStop is a hand-engineered policy that selects the *Jump* action, then stops as quickly as possible using actions $\{RollLeft, RollRight\}$. It utilizes a hand-engineered function *grounded* that checks if the agent is in contact with a surface.

$$\pi(s) \leftarrow \begin{cases} Jump & \text{if first time step for policy } \pi \\ NoAction & \text{if not } grounded(s) \\ RollLeft & \text{if } \dot{x} > stopVel \text{ and } grounded(s) \\ RollRight & \text{if } \dot{x} < -stopVel \text{ and } grounded(s) \\ \perp & \text{if } |\dot{x}| < stopVel \text{ and } grounded(s) \end{cases} \tag{5.5}$$

The stopping action is required in order to identify the ending segment where the agent comes to a complete stop. This ensures the planner can search any forward transitions from the ending segment.

Without stopping, the agent sometimes lands on segments at too high of a velocity to stop before rolling off the segment. These are not considered to be in the abstract segment state since the agent cannot utilize other forward transitions identified for the segment.

A side effect of the stopping behavior is that the planner will not find solutions that require jumping across multiple small segments without stopping. This could be solved in the future by adding another control policy, for example *JumpAndJump*.

5.3.2.3 FallAndStop

$FallAndStop$ is a hand-engineered policy that selects $NoAction$ until the agent rolls off the current surface, then stops as quickly as possible using actions $\{RollLeft, RollRight\}$.

$$\pi(s) \leftarrow \begin{cases} NoAction & \text{if } grounded(s) \text{ and on initial surface} \\ NoAction & \text{if not } grounded(s) \\ RollLeft & \text{if } \dot{x} > stopVel \text{ and } grounded(s) \text{ and not on initial surface} \\ RollRight & \text{if } \dot{x} < -stopVel \text{ and } grounded(s) \text{ and not on initial surface} \\ \perp & \text{if } |\dot{x}| < stopVel \text{ and } grounded(s) \text{ and not on initial surface} \end{cases} \tag{5.6}$$

The stopping action is required for the same reasons as $JumpAndStop$ and has similar limitations.

5.3.2.4 RollUp

$RollUp(x_{target}, y_{target}, \dot{x}_{target})$ is a hand-engineered policy that attempts to roll up onto a nearby surface located at (x_{target}, y_{target}) at velocity $\dot{x}_{target}$ using actions $\{RollLeft, RollRight\}$.

$$\pi(s) \leftarrow \begin{cases} RollLeft & \text{if } \dot{x} > \dot{x}_{target} \\ RollRight & \text{if } \dot{x} < \dot{x}_{target} \\ \perp & \text{if } \dot{x}_{target} > 0 \text{ and } x > x_{target} \text{ and } y < y_{target} \text{ and } grounded(s) \\ \perp & \text{if } \dot{x}_{target} < 0 \text{ and } x < x_{target} \text{ and } y < y_{target} \text{ and } grounded(s) \end{cases} \tag{5.7}$$

5.4 Planning

The high level AGPlan planning algorithm is shown in Algorithm 9. The planner explores space to construct an abstract model, then searches for a solution using the model. If a solution is not found, the planner continues to alternate refining the model and searching until an abstract plan is found or a timeout occurs.

Algorithm 9 AGPlan$\langle \mathcal{S}, \mathcal{A}, \mathcal{G}, \Pi, s_{init} \rangle$

```
1: $\mathcal{R} \leftarrow$ InitRegions$(\mathcal{S}, \mathcal{A}, \Pi)$
2: $s \leftarrow s_{init}$
3: $R \leftarrow$ Region$(s)$
4: $model \leftarrow$ InitModel$(\mathcal{S}, \mathcal{A}, \mathcal{G}, \Pi, \mathcal{R})$
5: while not $timeout$ do
6:     Explore$(R, model)$
7:     $\varphi \leftarrow$ Search$(s, \mathcal{G}, model)$
8:     if $\varphi \neq \varnothing$ then
9:         return $\varphi$
10: return $\varnothing$
```

5.4.1 Abstract States

The planner identifies a set of regions $\mathcal{R}$ in the state space such that any state in a region can be reached from any other state in the region by applying a single control policy. Each region defines an abstract state.

Let Region : $\mathcal{S} \mapsto \mathcal{R} \cup \bot$ specify the region, if any, containing a given state. (For simplicity this assumes regions do not overlap, which is not true in general. However, it is straightforward to extend support for overlapping regions.)

The task of identifying regions is highly specific to the state space and the available control policies. Regions may be found using hand-engineered analysis or may be identified using learned functions of the control policies.

It is not necessary to identify all regions during initialization; indeed, it is likely infeasible in large state spaces. Regions can be identified incrementally as reachable states are explored.

For GF, abstract states consist of flat surfaces on top of obstacles where the circle can roll and is not in collision with any obstacle. We refer to these regions as *segments*, denoted by the tuple $\langle x_{from}, x_{to}, y \rangle$.

Each segment region is further bounded by the range of x velocities achievable using the *RollTo* policy starting from zero velocity at any point on the segment. y velocities are limited to $|\dot{y}| < \dot{y}_{grounded}$ which ensures the agent is in contact with the surface and can accelerate or decelerate.

5.4.2 Abstract Model

The planner incrementally constructs an abstract model containing the regions along with a set of potentially useful trajectories for each region. Trajectories are considered useful if they achieve a goal, or result in a transition between regions.

Algorithm 10 shows the procedure for exploring and extending the model starting from region R. The planner first explores R, then recursively explores all regions reachable from R. A region is not explored until the planner finds a way to reach it.

For each exploration step, the planner selects a starting state s in the current region R and selects a control policy $\pi = \pi_i(\theta)$ for some applicable $\pi_i \in \Pi$ with parameters θ. Policy selection may be entirely random, or may be biased toward potentially more useful policies based on knowledge of the problem domain.

The planner then simulates applying policy π from s and evaluates the resulting trajectory τ. If the trajectory achieves any goals or results in a transition to another region, the tuple $\langle \tau, \pi \rangle$ is saved with the region.

Algorithm 10 Explore$\langle R, model \rangle$

```
1: $model.Regions \leftarrow model.Regions \cup R$
2: loop GetEffort$(R, model)$ times
3:     $s \leftarrow$ SelectState$(R, model)$
4:     $\pi(\theta) \leftarrow$ SelectPolicy$(s, model)$
5:     $\tau \leftarrow$ SimulatePolicy$(s, \pi(\theta))$
6:     $s' \leftarrow$ Last$(\tau)$
7:     $R' \leftarrow$ Region$(s')$
8:     if Goals$(\tau) \neq \varnothing$ or $R' \neq R$ then
9:         $model.Trajectories(R).Add(\langle \tau, \pi(\theta) \rangle)$
10:    if $R' \neq R$ and $R' \notin model.Connections(R)$ then
11:        $model.Connections(R).Add(R')$
12: for $R' \in model.Connections(R)$ do
13:    Explore$(R', model)$
```

The number of iterations controls the level of exploration effort for each region and can allow the planner to focus on the most relevant regions.

5.4.3 Engineered Exploration in Geometry Friends

GF provides a physics simulator which is used to determine the result of applying a control policy from a state. The game provides a simulation state for the current state to the agent at each time step. Simulation states can be saved and copied, but cannot be created for arbitrary states. As a consequence, simulation is limited to a forward search of states reachable from the agent starting state.

For the initial implementation of the planner for GF, the state and policy sampling in lines 3-4 of Algorithm 10 uses the following highly-engineered strategy that combines hand-coded geometric analysis with random exploration.

5.4.3.1 Potential Segment Transitions

For each pair of segments (s_{from}, s_{to}), a set of possible transitions is identified.

Potential jump transitions are identified by evaluating the predicted trajectory of jumping from s_{from} at $(x, \dot{x})$ for a range of x and $\dot{x}$ values. A trajectory is assumed to be a simple parabolic path determined by the starting location, velocity, and gravity in the y direction. If the trajectory intersects s_{to} then a potential transition is created using a composite policy of $[\mathit{RollTo}(x, \dot{x}), \mathit{JumpAndStop}]$.

Potential roll-off transitions are identified by evaluating the predicted trajectory of rolling off either end of s_{from} for a range of $\dot{x}$ values. If the trajectory intersects s_{to} then a potential transition is created using a composite policy of $[\mathit{RollTo}(x_{end}, \dot{x}), \mathit{FallAndStop}]$.

Potential roll-up transitions are identified by checking if either end of s_{to} is near s_{from} with $0 < y_{to} - y_{from} < y_{maxRollUp}$. If so, then potential transitions for a range of $\dot{x}$ values are created with policy $\mathit{RollUp}(x_{to}, y_{to}, \dot{x})$ where (x_{to}, y_{to}) is the nearest corner of s_{to}.

The simplistic predicted trajectories for jump and roll-off transitions completely ignore obstacles and the potential transitions often cannot reach s_{to}.

5.4.3.2 Potential Goal Collects

Potential jumps to collect goals are identified in a similar way as jump transitions. For every pair (s, g_i) of segment s and goal g_i, potential jump collects are identified by evaluating the predicted trajectory of jumping from $(x, \dot{x})$ for a range of x and $\dot{x}$ values. If the trajectory intersects g_i then a potential collect is created using a composite policy of $[\mathit{RollTo}(x, \dot{x}), \mathit{JumpAndStop}]$.

Likewise, potential roll-offs to collect goals are identified by evaluating the predicted trajectory of rolling off either end of s for a range of $\dot{x}$ values. If the trajectory intersects g_i then a potential collect is created using a composite policy of $[\mathit{RollTo}(x_{end}, \dot{x}), \mathit{FallAndStop}]$.

If a goal can be reached by directly rolling to a point x on s, then a roll collect is created with policy $RollTo(x, 0)$. In this case the collect is guaranteed to succeed.

5.4.3.3 Exploring Jumps

Using only the potential transitions and collects identified above, the planner will fail to find many solutions. Since the potential trajectories assume an obstacle-free environment the planner will not identify many useful policies, for example those that require collisions with obstacles.

Random exploration is required to find useful policies not identified by the initial analysis. This is accomplished using exploring jumps for every segment s over the full range of possible $(x, \dot{x})$ values with a composite policy of $[RollTo(x, \dot{x}), JumpAndStop]$. Exploring jumps are randomly selected without predicting the outcome and can eventually find transitions and collect goals that otherwise would not be found.

When exploring a segment, the planner randomly selects from the potentially useful policies and exploring jumps identified in the initial analysis. It then simulates executing each selected policy starting from a previously recorded simulation state in the segment. The first time the planner reaches a segment, the simulation state is saved with the segment to enable subsequent exploration from that point.

5.4.4 Search

Heuristic search is used to find a solution in the form of an abstract plan as shown in Algorithm 11.

Nodes in the search correspond to $\langle s, G \rangle$ where $s \in \mathcal{S}$ and G is the set of remaining goals. When a node is expanded, successor nodes are generated for each saved trajectory $\langle \tau_t, \pi_t \rangle$ associated with the node region. For each successor the planner must first find a local trajectory from the current node state s to the start of

Algorithm 11 $\textsc{Search}\langle s_{init}, \mathcal{G}, model\rangle$

```
1: $openList \leftarrow \textsc{EmptyPriorityQueue}()$
2: $h \leftarrow \textsc{CalculateHeuristic}(s_{init})$
3: $openList.Insert(\textsc{RootNode}(s_{init}, \mathcal{G}, 0, h))$
4: while not $timeout$ and not $openList.IsEmpty()$ do
5:     $node \leftarrow openList.Pop()$
6:     if $node.Goals = \varnothing$ then
7:         return $\textsc{CreatePlan}(node)$
8:     $s \leftarrow node.State$
9:     $R \leftarrow \textsc{Region}(s)$
10:    for $\langle \tau_t, \pi_t\rangle \in model.Trajectories(R)$ do
11:        $\langle \tau_s, \pi_s\rangle \leftarrow \textsc{GetTrajectory}(s, \textsc{First}(\tau_t))$
12:        $\tau \leftarrow \tau_s + \tau_t$
13:        $\pi \leftarrow \textsc{CompositePolicy}(\pi_s + \pi_t)$
14:        $s' \leftarrow \textsc{Last}(\tau)$
15:        $G' \leftarrow \mathcal{G} - \textsc{Goals}(\tau)$
16:        $cost \leftarrow node.Cost + \textsc{Cost}(\tau)$
17:        $h' \leftarrow \textsc{CalculateHeuristic}(s')$
18:        $succ \leftarrow \textsc{ChildNode}(node, \pi, s', G', cost, h')$
19:        $openList.Insert(succ)$
20: return $\varnothing$
```

the saved trajectory $\textsc{First}(\tau_t)$. This is guaranteed possible by the region condition that every state is reachable from every other state using a single policy.

The successor node then combines the local and saved trajectories, and removes any remaining goals achieved by the combined policy.

A complete solution is found when the search selects a node with no remaining goals. An abstract plan φ consisting of a sequence of states and control policies $[\langle s_0, \pi_0\rangle \dots \langle s_n, \pi_n\rangle]$ is obtained by tracing the ancestors of the solution node back to the initial node.

Some implementation details are not shown for brevity. An important consideration is that equality checking for nodes is only approximate. This controls the branching factor by limiting the number of nodes for each region. For example, the number of nodes can be minimized by specifying $\langle s, G\rangle \approx \langle s', G'\rangle \iff \textsc{Region}(s) = \textsc{Region}(s') \wedge G = G'$. This reduces the search complexity but produces less optimal solutions since it ignores differences in cost-to-go for node states in the same region. This trade-off can be balanced by partitioning large regions into smaller sub-regions for the purpose of node equality checking.

As usual in heuristic search, nodes in the open list may be sorted using any weighted combination of the cost-to-reach (g) and cost-to-go (h) estimates.

The search also keeps track of the best node found so far, i.e. one with the fewest remaining goals and the lowest cost. This node may be used to generate a partial plan if a complete plan is not found.

For GF a variety of heuristic functions $h(s)$ have been tested but simply using $h(s) = 0$ is an effective choice, which essentially reduces planning to breadth-first search based on cost. Exploration in this instance requires on the order 10^2 more time than planning so it is not worth computing a more informative heuristic. Nodes in the search correspond to $\langle R, C\rangle$ where R is a segment region and C is the set of remaining collectibles.

5.4.4.1 Risk Avoidance

Sometimes the planner finds policies that appear useful in simulation but are unreliable during plan execution. An example in GF is a plan that requires the agent to bounce off a corner. In simulation the behavior is deterministic and reliable, but during game execution small variations in the agent position and velocity result in widely divergent trajectories after hitting the corner. At best, this wastes time re-planning from an unexpected state. At worst, it leaves the agent stuck in a dead-end unable to finish the problem. Thus, trajectories that hit corners have a much higher risk of failure.

To address this problem, a function GetRiskPenalty(π, τ) evaluates the risk of failure for a given policy trajectory and assigns a cost penalty based on the risk. During search the Cost(τ) function includes the risk penalty so the agent tends to avoid risky trajectories.

Using the risk penalty the GF agent is less likely to fail during plan execution and is better at avoiding dead-ends. However, the solutions found tend to be slower than when it is not used. Furthermore the magnitude of the risk penalty is a parameter that must be tuned.

5.5 Execution

The high level AGAgent control algorithm that interleaves planning and execution is shown in Algorithm 12.

Algorithm 12 AGAgent$\langle \mathcal{S}, \mathcal{A}, \mathcal{G}, \Pi \rangle$

```
1:  $model \leftarrow$ InitModel($\mathcal{S}, \mathcal{A}, \mathcal{G}, \Pi, \mathcal{R}$)
2:  while not IsSolved() do
3:      $s \leftarrow$ GetCurrentState()
4:      $G \leftarrow$ GetRemainingGoals()
5:      $\varphi \leftarrow$ AGPlan($s, G, model$)
6:      for $\langle s_i, \pi_i \rangle \in \varphi$ do
7:          InitPolicy($\pi_i, s_i$)
8:          while not IsFinished($\pi_i$) do
9:              $s \leftarrow$ GetCurrentState()
10:             $a \leftarrow$ GetAction($\pi_i, s$)
11:             SelectAction($a$)
12:         if Failed($\pi_i$) then
13:             Abort($\varphi$)
```

Once an initial plan has been found, the agent starts execution by selecting actions using the first policy in the plan until the policy completes. If the resulting state satisfies the preconditions for the next policy in the plan, the agent proceeds with the next policy and so on until the entire plan has been completed.

If at any point a policy fails to complete with the expected preconditions for the next policy, the plan is considered to have failed and the planner replans from the current state, resulting in further exploration and a new search.

Any policy failures during execution are recorded so that the policy is not selected in subsequent searches. This prevents the agent from repeatedly attempting a policy that succeeds in simulation but fails during actual execution.

If the planner fails to find a complete solution within a specified timeout period, the agent proceeds with the best partial plan found so far.

The circle agent for GF is implemented using AGAgent.

5.5.1 Optimizations

Using the provided control policies the GF agent is able to find good quality plans and solve all levels tested. However, execution of the resulting plans is significantly improved with minor modifications.

5.5.1.1 Early Policy Termination

In general, when executing a plan, the current policy can be terminated as soon as the goals for the policy have been achieved and the preconditions for starting the next policy are satisfied, i.e. the agent is in the attractor region for the next control policy.

The GF agent slows down and stops between every step in the initial plan since the *JumpAndStop* and *FallAndStop* policies bring the agent to a complete stop. Stopping is useful during planning because it ensures the agent can come to a full stop on a segment and therefore can reach any other point on the segment. Yet during plan execution, stopping is unnecessary once we reach the target segment. As a result, the agent can proceed immediately with the next policy when it reaches the next segment, without coming to a stop. The resulting motion is much faster and smoother.

5.5.1.2 Redundant Policies

If a plan has two policies in sequence and executing the second policy achieves the same goals as the first policy, the first policy is redundant and can be removed from the plan.

In GF this occurs when the agent uses *RollTo* to collect a goal on the current segment, followed by a *RollTo* for a jump or fall transition. Eliminating the first *RollTo* allows the agent to immediately accelerate for the jump or fall, while still collecting the goal.

5.6 Results

The performance of AGAgent was evaluated on all Geometry Friends levels from past competitions. Each competition has 10 levels and results are averaged over 10 runs per level. Results for the two best-performing agents from prior competitions are provided for comparison in Figure 5.2.

AGAgent is able to solve all levels for every Geometry Friends competition and outperforms all agents from prior competitions except one.

The KITAgent planner [43] achieves excellent performance by predicting accurate trajectories using a fast hand-engineered forward simulation that emulates the actual game physics. The agent initializes an exhaustive set of predicted trajectories that are used directly for planning, enabling it to find efficient solutions quickly. No further exploration is performed after initialization.

Competition	Goals	RRTAgent2017		KITAgent2017		AGAgent	
		Goals	Time	Goals	Time	Goals	Time
2013	28	19.8	848.4	**28**	**199.1**	27.3	320.5
2014	28	14.8	547.1	**27.8**	**189.4**	27.4	292.5
2015	30	15	562.8	**29.6**	**263.6**	27.2	367.3
2016	35	17	973.9	**34**	**432.1**	32.9	601.9
2017	26	21.2	726.2	**25.4**	**226.5**	**25.4**	280.3
2019	27	22	736.6	24	**237.3**	**26.9**	342.3

Figure 5.2: Geometry Friends Results

KITAgent solves most levels faster than AGAgent. However, AGAgent can solve some levels not possible with KITAgent. AGAgent uses the native physics simulation provided by the game and can accurately predict the outcome of any control policy from any state, whereas the KITAgent simulation is accurate only for trajectories with no more than one obstacle collision. As a result AGAgent can use a richer set of control policies applicable to a larger variety of problems, at the cost of additional computation time. Furthermore, AGAgent utilizes random sampling of control policies during exploration and can find solutions that would otherwise not be considered. KITAgent does not perform random sampling at all. Figure 5.3 shows a difficult level solvable by AGAgent but not KITAgent. The tight passages and irregular surfaces require a series of collisions to navigate and the hand-engineered simulation in KITAgent does not find a feasible path.

In addition to the KITAgent, performance is also compared with the RRTAgent. The RRTAgent planner [55] is based on RRT with the STP approach using 3 simple skills. While it manages to collect a majority of goals, it is the slowest of the three planners.

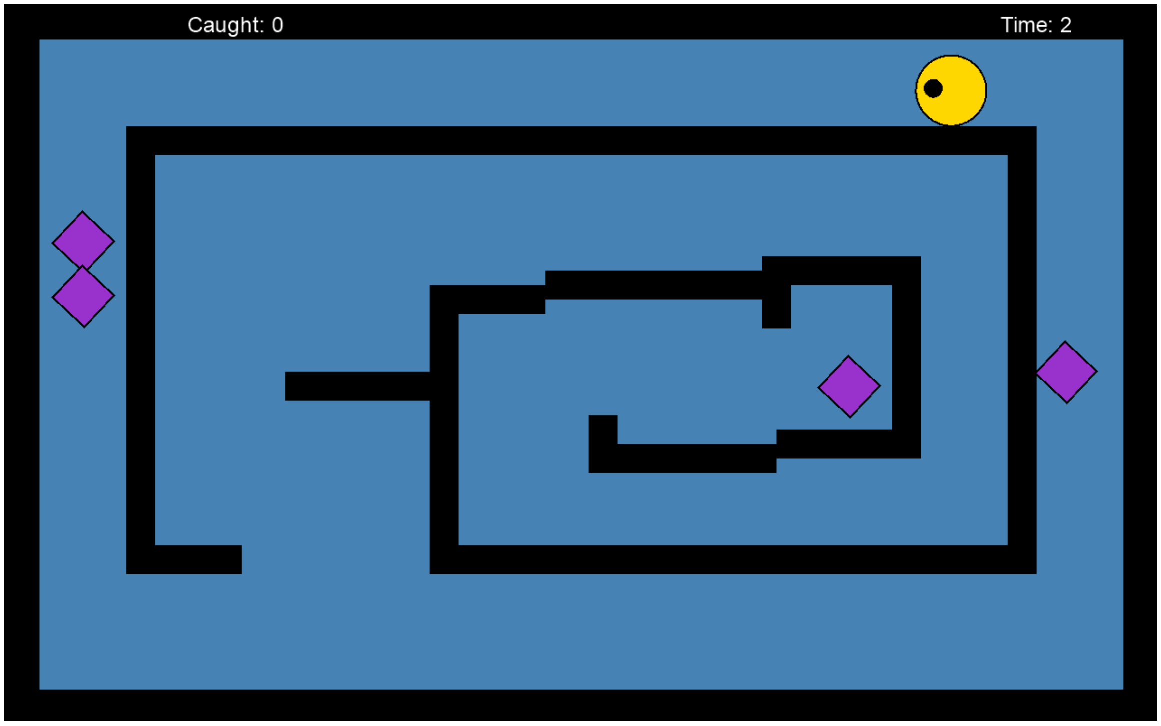


Figure 5.3: 2016 Level 1

5.7 Conclusions and Further Work

Initial experiments using standard sampling-based methods for kinodynamic planning performed poorly in the GF domain. RRT-based planning using randomized control strategies and macro-actions could find solutions only for the very simplest problems. A key problem is the lack of an effective distance metric, leading the planner to make poor decisions when selecting nodes in the tree. Another problem is the lack of an effective steering mechanism to approach arbitrary sampled points in free space.

Abstraction of the state space appears essential for this domain. Indeed, all of the competitive entries in GF competitions from 2013-2019 have utilized similar abstractions based on identifying flat navigable surfaces.

The related works cited in Section 2.7 generally rely on abstractions based on simple geometric decompositions of the workspace using triangulation or grid-based approaches. These approaches are effective for general 2D and 3D agent navigation when the planning state space is uniform, i.e. the agent has the same control actions available at any point in free space and is equally maneuverable in all directions.

These conditions do not hold in the GF domain. The agent has direct control over linear acceleration only when in stable contact with a surface and cannot accelerate or jump when in free space not in contact with a surface. Furthermore, the agent is subject to gravity in one dimension resulting in a non-symmetric state space for planning.

An analysis of the 60 competition GF levels shows that the volume of useful abstract states comprises only 4.5×10^{-5} of the state space on average. Thus, naive standard planning methods spend 99.99% of their time exploring states that are of little use in this domain.

In contrast, the work presented here offers a general approach that defines abstract states in relation to the available control policies rather than in terms of geometric, domain-specific attributes. By focusing on identifying abstract states and finding transitions between them, the planner is able to find good quality plans much more quickly.

The competition agent described here uses a highly-engineered exploration strategy tailored to the problem domain that does not take into account obstacles. To reduce this need of hand-engineering, the following chapters will investigate techniques that use sampling and machine learning techniques to provide efficient exploration and facilitate the automatic formation of abstract states and useful control policies. Chapter 6 presents an exploration strategy that uses predictive models to avoid simulating policies that are unlikely to succeed based on the full problem state including obstacles. In addition a pure random exploration strategy is implemented for comparison with the engineered strategy.

Lastly, this planner relies on a domain-specific hand-engineered identification of the abstract states. Chapter 7 proposes a more general sampling-based approach for identifying abstract states.

CHAPTER 6

PREDICTIVE MODELS FOR EXPLORATION

6.1 Introduction

The GF competition agent described in Chapter 5 uses a highly-engineered exploration strategy tailored to the problem domain in order to sample policies for simulation.

While the implemented approach works reasonably well in practice, it has some key shortcomings:

- The engineered analysis for selecting policies to sample and simulate is naive and ignores obstacles. This leads to wasted time simulating many policies that have little hope of success because the analysis makes false assumptions about the obstacle configuration in the environment.
- The engineered analysis is limited and does not identify feasible policies with trajectories that require collisions with obstacles.
- The engineered analysis itself is highly domain-specific and cannot be generalized.

This chapter seeks to address these shortcomings in two ways.

The first question explored is, can we learn to predict where a control policy is likely to be useful within an abstract state? If so, can we use these predictions to improve planning efficiency?

The second question explored is, how does a completely random exploration strategy compare with the highly engineered strategy? Can predictive models improve a random exploration strategy as well?

6.2 Predictive Models

The first question considered is whether we can accurately predict when control policies will be useful in a range of situations. To investigate this we evaluate the following 5 models in this section:

- The Predict Goal model predicts whether applying a policy $\pi(\theta)$ from state s can reach a specified goal g
- The Predict Region model predicts whether applying a policy $\pi(\theta)$ from state s can reach a specified target region R'
- The Predict Useful model predicts whether applying a policy $\pi(\theta)$ from state s can reach any goal or region not equal to the starting region
- The Predict Region-to-Goal model predicts whether applying a policy π from region R can reach a specified goal g
- The Predict Region-to-Region model predicts whether applying a policy π from region R can reach a specified target region R'

6.2.1 Model Architecture

Prediction of a control policy outcome is highly dependent on the location of the agent, the goal, and the obstacles in the surrounding enviroment. Convolution using input images of the workspace offers a promising approach to capture these dependencies since physical workspaces are intrinsically 2D or 3D and the system behavior is location-invariant. Using convolution enables a network to learn features and spatial relationships that may occur anywhere in the workspace and offers a natural way to represent obstacles and physically situated parameters such as the agent and goal locations.

All experiments used convolutional neural networks to learn the prediction functions [38]. Each of the networks used a similar design, with a multi-channel input image and a vector of input parameters.

6.2.2 Training Data

To obtain training data, a set of 10000 random GF levels were generated. Each level contains between 2 and 4 platforms, each with a random size and location. In addition, each level contains 100 randomly located goals.

Training samples were created using Algorithm 13. Random policy instances were simulated for every abstract state region (segment) in each level. Each sample records the start region and end region after applying the policy, as well as any goals reached by the policy. An average of 1200 samples per level were created for a total of approximately 1.2×10^7 samples.

Algorithm 13 GeneratePolicySamples$\langle levels, samplingDensity\rangle$

```
1: for $level \in levels$ do
2:     for $R \in level.Regions$ do
3:         loop GetSampleCount($R, samplingDensity$) times
4:             $s \leftarrow$ SelectRandomState($R$)
5:             $\pi, \theta \leftarrow$ SelectRandomPolicy($s$)
6:             $\tau \leftarrow$ SimulatePolicy($s, \pi(\theta)$)
7:             $R' \leftarrow$ Region(Last($\tau$))
8:             SaveSample($s, \pi, \theta, R, R'$, Goals($\tau$))
```

6.2.3 Predict Goal Model

This model predicts whether applying an instance of a control policy starting in a specified state will result in the agent reaching a specified goal.

The prediction function has the following form:

$$\textsc{PredictGoal}(s, \pi, \theta, goalLocation) \mapsto [0..1]$$

where an output > 0.5 predicts the policy will achieve the specified goal.

Figure 6.1 shows the high-level network architecture. A 9-channel input image is processed by 4 stacked convolutional layers using LeakyReLU($\alpha = 0.2$) activation with a stride of 2, kernel size 3, and 32 filters each. A batch normalization layer is included after each of the convolutional layers. The convolutional output layer is then flattened and concatenated with a vector input, followed by a fully-connected hidden layer of 1000 nodes. Lastly dropout (rate=0.2) is applied and connected to a single sigmoid output node. Training uses a binary cross-entropy loss function with the Adam optimizer with learning rate 1×10^{-3}.

The convolutional input images correspond to a rescaled global image of the 2D workspace as shown in Figure 6.2. Each image input channel encodes an input parameter in the corresponding spatial location in the workspace image.

The 9 input channels assign the workspace pixel values as follows:

- Obstacles: 1 if occupied by an obstacle, else 0
- Agent Location: 1 if occupied by the agent, else 0
- Goal Location: 1 if occupied by the goal, else 0
- Agent X Velocity: $\dot{x}$ if occupied by the agent, else 0
- Agent Y Velocity: $\dot{y}$ if occupied by the agent, else 0
- JumpAndStop indicator: 1 if occupied by the agent and $\pi = JumpAndStop$, else 0
- FallAndStop indicator: 1 if occupied by the agent and $\pi = FallAndStop$, else 0
- RollUp indicator: 1 if occupied by the agent and $\pi = RollUp$, else 0
- RollUp X Velocity: $\dot{x}_{target}$ if occupied by the agent and $\pi = RollUp$, else 0

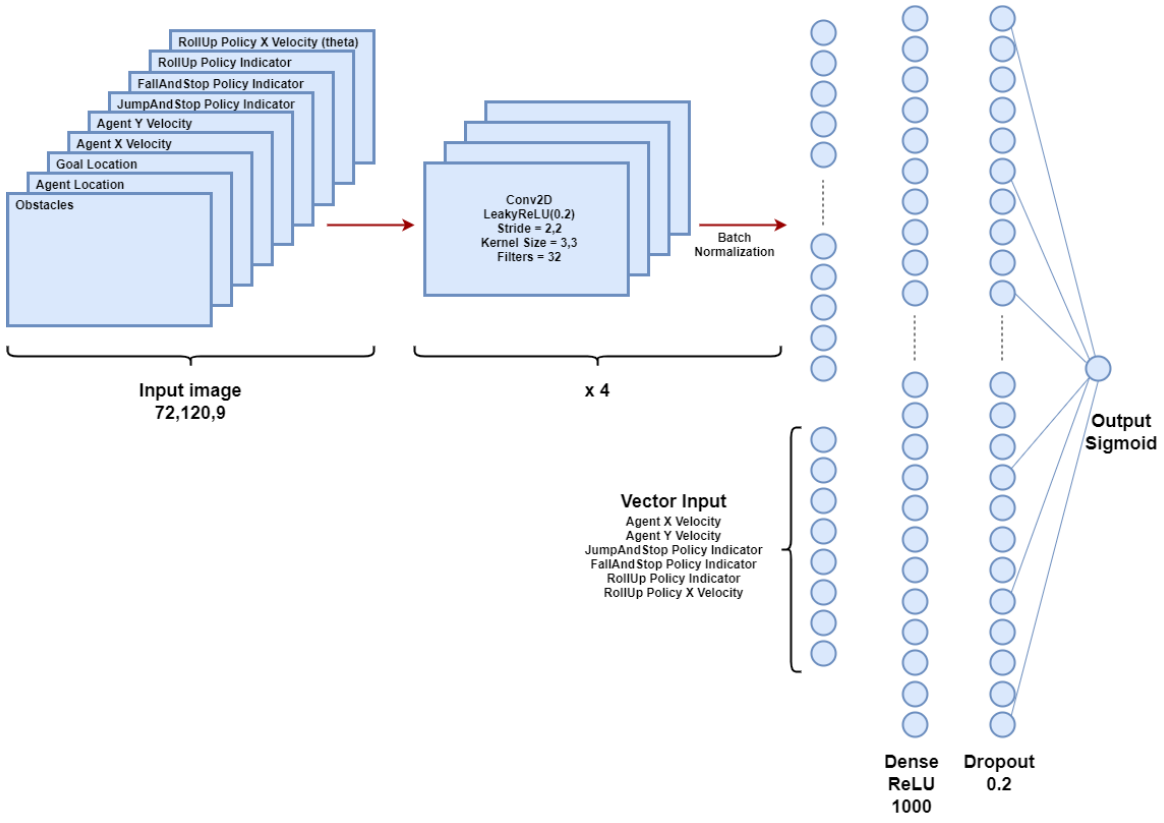
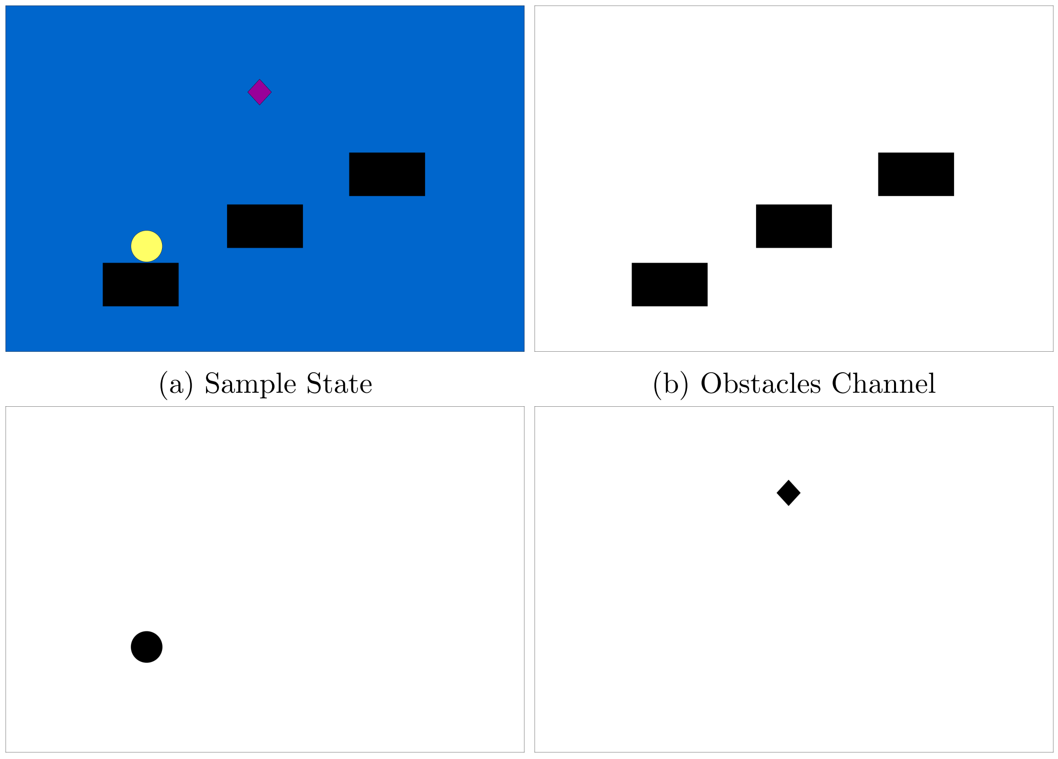


Figure 6.1: Predict Goal Network Architecture

Network training consumes an infinite random sequence of goal prediction samples drawn from the training policy samples using Algorithm 14. The sequence ensures uniform coverage and alternates positive and negative training samples to avoid the negative sample bias which otherwise occurs since there are far more ways to fail to achieve a goal than to succeed.

The network implementation used Keras and Tensorflow [1] [11]. The network architecture and parameters were selected on the basis of limited informal testing and have not been otherwise tuned.


(a) Sample State (b) Obstacles Channel

(c) Agent Channel (d) Goal Channel

Figure 6.2: Convolutional Image Channels

6.2.3.1 Results

The network was trained using 9900 of the training levels with 100 levels used only for evaluation. As shown in Figure 6.3, the model reaches 96.6% accuracy over 3000 training epochs.

It is worth noting there is no sign of overfitting for this model as the evaluation accuracy closely tracks training accuracy. This appears to be a result of the extremely large pool of training samples and suggests the batch normalization and dropout layers in the network may not be needed.

Algorithm 14 GenerateGoalSamples⟨*levels*⟩

```
1:  $isPositive \leftarrow true$
2:  loop forever
3:      $level \leftarrow \textsc{SelectRandom}(levels)$
4:      $goal \leftarrow \textsc{SelectRandom}(level.Goals)$
5:      if $isPositive$ then
6:          $\sigma \leftarrow \textsc{SelectRandom}(\sigma \in level.PolicySamples : goal \in \sigma.Goals)$
7:      else
8:          $\sigma \leftarrow \textsc{SelectRandom}(\sigma \in level.PolicySamples : goal \notin \sigma.Goals)$
9:      yield $\textsc{GoalSample}(\sigma.state, \sigma.\pi, \sigma.\theta, goal, isPositive)$
10:     $isPositive \leftarrow \neg isPositive$
```

6.2.4 Predict Region Model

This model predicts whether applying an instance of a control policy starting in a specified state will result in the agent reaching a specified target region.

The prediction function has the following form:

$$\textsc{PredictRegion}(s, \pi, \theta, regionLocation) \mapsto [0..1]$$

where an output > 0.5 predicts the policy will achieve the specified region.

The network architecture is the same as the Predict Goal network shown in Figure 6.1 except the Goal Location channel input is replaced by a Region Location channel:

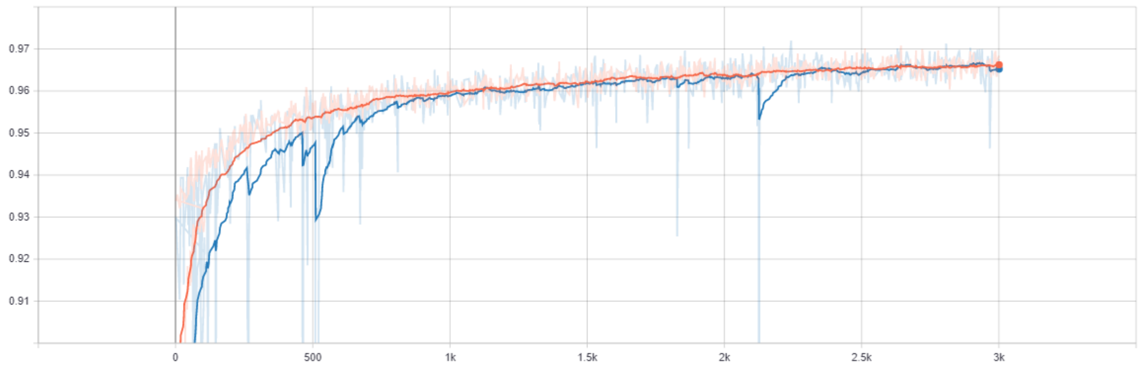


Figure 6.3: Predict Goal Network Training Accuracy (Orange = training data, Blue = evaluation data)

- Region Location: 1 if occupied by the target region, else 0

Network training is similar to the Predict Goal model and consumes samples generated by Algorithm 15.

Algorithm 15 GenerateRegionSamples⟨*levels*⟩

```
1: $isPositive \leftarrow true$
2: loop forever
3:     $level \leftarrow \textsc{SelectRandom}(levels)$
4:     $R \leftarrow \textsc{SelectRandom}(level.Regions)$
5:     if $isPositive$ then
6:         $\sigma \leftarrow \textsc{SelectRandom}(\sigma \in level.PolicySamples : \sigma.R \neq R \wedge \sigma.R' = R)$
7:     else
8:         $\sigma \leftarrow \textsc{SelectRandom}(\sigma \in level.PolicySamples : \sigma.R \neq R \wedge \sigma.R' \neq R)$
9:     yield $\textsc{RegionSample}(\sigma.state, \sigma.\pi, \sigma.\theta, R, isPositive)$
10:    $isPositive \leftarrow \neg isPositive$
```

6.2.4.1 Results

As shown in Figure 6.4, the model reaches 94.8% accuracy over 3500 training epochs.

In contrast to the Predict Goal model, this model shows some overfitting as the evaluation accuracy trails training accuracy. This reflects the smaller pool of training samples since there are only 3 to 5 target regions on average per level vs. 100 goals per level.

6.2.5 Predict Useful Model

This model predicts whether applying an instance of a control policy starting in a specified state will be useful, i.e. will it result in the agent reaching *any* goal or *any* region not equal to the starting region.

The prediction function has the following form:

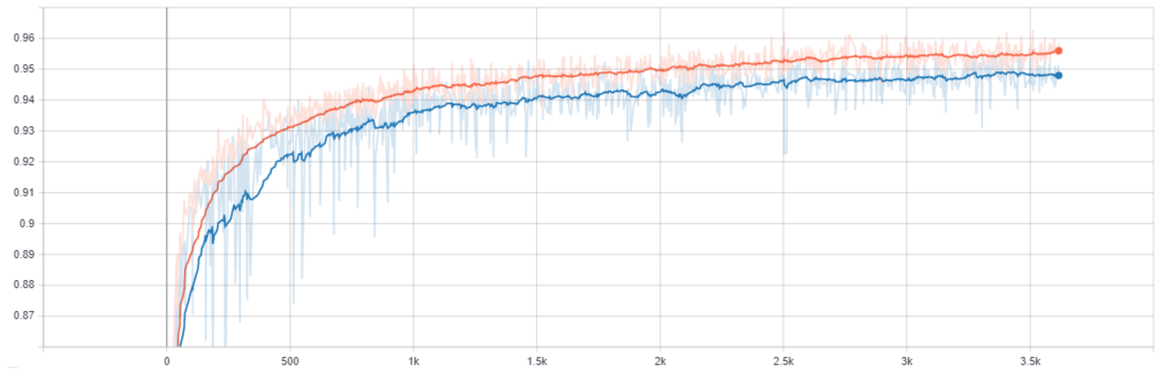


Figure 6.4: Predict Region Network Training Accuracy (Orange = training data, Blue = evaluation data)

$$\textsc{PredictUseful}(s, \pi, \theta, goalLocations, regionLocations) \mapsto [0..1]$$

where an output > 0.5 predicts the policy will reach at least one of the goals or regions.

The network architecture is the same as the Predict Goal network shown in Figure 6.1 except the Goal Location channel input is replaced by 2 channels:

- Goal Locations: 1 if occupied by any goal, else 0
- Region Locations: 1 if occupied by any region, else 0

Network training is similar to the Predict Goal model and consumes samples generated by Algorithm 16. One additional training parameter required for this model is *goalsPerSample*. The generated training levels contain 100 goals per level, which is far more than typical GF levels that contain 1-5 goals. To compensate for this, the generator randomly selects a subset of 5 goals from the 100 available goals each time a sample is generated.

6.2.5.1 Results

As shown in Figure 6.5, the model reaches 93.6% accuracy over 2400 training epochs.

Algorithm 16 $\textsc{GenerateUsefulSamples}\langle levels, samplesPerGoal\rangle$

```
1: $isPositive \leftarrow true$
2: loop forever
3:     $level \leftarrow \textsc{SelectRandom}(levels)$
4:     $G \leftarrow \textsc{SelectRandomSubset}(level.Goals, samplesPerGoal)$
5:     if $isPositive$ then
6:         $\sigma \leftarrow \textsc{SelectRandom}(\sigma \in level.PolicySamples : \sigma.R \neq \sigma.R' \vee |\sigma.Goals \cap G| > 0)$
7:     else
8:         $\sigma \leftarrow \textsc{SelectRandom}(\sigma \in level.PolicySamples : \sigma.R = \sigma.R' \wedge |\sigma.Goals \cap G| = 0)$
9:     yield $\textsc{UsefulSample}(\sigma.state, \sigma.\pi, \sigma.\theta, G, level.Regions, isPositive)$
10:    $isPositive \leftarrow \neg isPositive$
```

As with the Predict Goal model, this model shows no overfitting as the evaluation accuracy closely tracks training accuracy. This reflects the vastly expanded training diversity introduced by selecting a random goal subset for each sample.

6.2.6 Predict Region-to-Goal Model

Up to this point the models have made predictions given a specific state s and policy instance $\pi(\theta)$. In contrast, this model makes a broader prediction whether

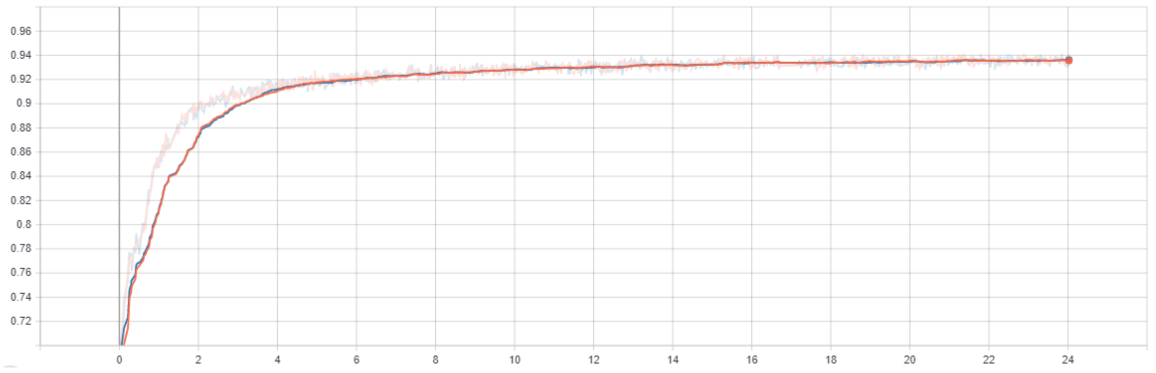


Figure 6.5: Predict Useful Network Training Accuracy (Orange = training data, Blue = evaluation data)

applying a control policy π with *any* θ starting from *any* state in a specified region will reach a specified goal.

The prediction function has the following form:

$$\textsc{PredictRegionToGoal}(regionLocation, \pi, goalLocation) \mapsto [0..1]$$

where an output > 0.5 predicts whether any instance of the policy can reach the goal starting from the specified region.

The network architecture is the same as the Predict Goal network shown in Figure 6.1 but uses the following image channels:

- Obstacles: 1 if occupied by an obstacle, else 0
- Starting Region Location: 1 if occupied by the starting region, else 0
- Goal Location: 1 if occupied by the goal, else 0
- JumpAndStop indicator: 1 if occupied by the starting region and $\pi = JumpAndStop$, else 0
- FallAndStop indicator: 1 if occupied by the starting region and $\pi = FallAndStop$, else 0
- RollUp indicator: 1 if occupied by the starting region and $\pi = RollUp$, else 0

Network training is similar to the Predict Goal model and consumes samples generated by Algorithm 17.

6.2.6.1 Results

As shown in Figure 6.6, the model reaches 95.1% accuracy over 1500 training epochs.

The model is more prone to overfitting since the number of combinations of starting regions and policy types is small compared to the possible number of starting states.

Algorithm 17 GENERATEREGIONTOGOALSAMPLES$\langle levels, \Pi\rangle$

```
1:  $isPositive \leftarrow true$
2:  loop forever
3:      $level \leftarrow$ SELECTRANDOM($levels$)
4:      $goal \leftarrow$ SELECTRANDOM($level.Goals$)
5:      $\pi \leftarrow$ SELECTRANDOM($\Pi$)
6:      if $isPositive$ then
7:          $\sigma \leftarrow$ SELECTRANDOM($\sigma \in level.PolicySamples : \sigma.\pi = \pi \wedge goal \in \sigma.Goals$)
8:      else
9:          $\sigma \leftarrow$ SELECTRANDOM($\sigma \in level.PolicySamples : \sigma.\pi = \pi \wedge goal \notin \sigma.Goals$)
10:     yield REGIONTOGOALSAMPLE($\sigma.R, \sigma.\pi, goal, isPositive$)
11:     $isPositive \leftarrow \neg isPositive$
```

6.2.7 Predict Region-to-Region Model

This model predicts whether applying a control policy π with *any* θ starting from *any* state in a specified region will reach a specified target region.

The prediction function has the following form:

$$\text{PREDICTREGIONTOREGION}(regionLocation, \pi, targetRegionLocation) \mapsto [0..1]$$

where an output > 0.5 predicts whether any instance of the policy can reach the target region from the starting region.

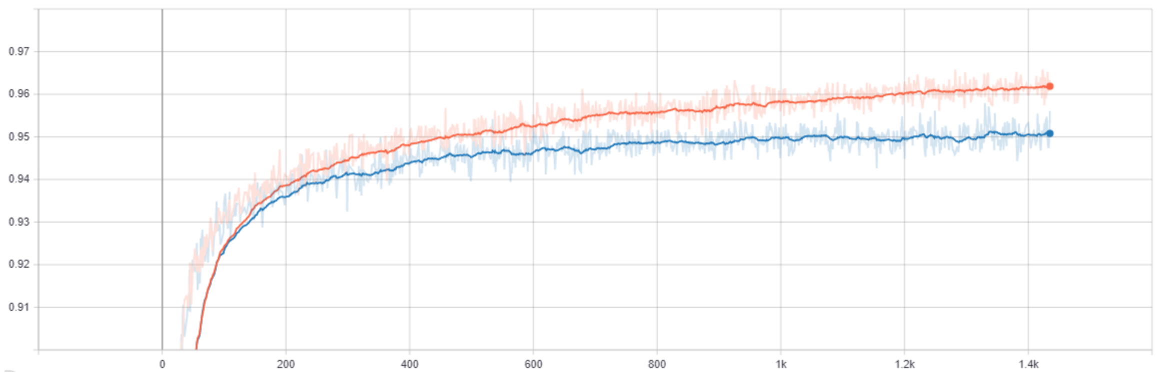


Figure 6.6: Predict Region-to-Goal Network Training Accuracy (Orange = training data, Blue = evaluation data)

The network architecture is the same as the Predict Goal network shown in Figure 6.1 but uses the following image channels:

- Obstacles: 1 if occupied by an obstacle, else 0
- Starting Region Location: 1 if occupied by the starting region, else 0
- Target Region Location: 1 if occupied by the target region, else 0
- JumpAndStop indicator: 1 if occupied by the starting region and $\pi = JumpAndStop$, else 0
- FallAndStop indicator: 1 if occupied by the starting region and $\pi = FallAndStop$, else 0
- RollUp indicator: 1 if occupied by the starting region and $\pi = RollUp$, else 0

Network training is similar to the Predict Goal model and consumes samples generated by Algorithm 18.

Algorithm 18 GenerateRegionToRegionSamples$\langle levels, \Pi \rangle$

```
1:  $isPositive \leftarrow true$
2:  loop forever
3:      $level \leftarrow$ SelectRandom($levels$)
4:      $R' \leftarrow$ SelectRandom($level.Regions$)
5:      $\pi \leftarrow$ SelectRandom($\Pi$)
6:      if $isPositive$ then
7:          $\sigma \leftarrow$ SelectRandom($\sigma \in level.PolicySamples : \sigma.\pi = \pi \wedge \sigma.R \neq R' \wedge \sigma.R' = R'$)
8:      else
9:          $\sigma \leftarrow$ SelectRandom($\sigma \in level.PolicySamples : \sigma.\pi = \pi \wedge \sigma.R \neq R' \wedge \sigma.R' \neq R'$)
10:     yield RegionToRegionSample($\sigma.R, \sigma.\pi, R', isPositive$)
11:     $isPositive \leftarrow \neg isPositive$
```

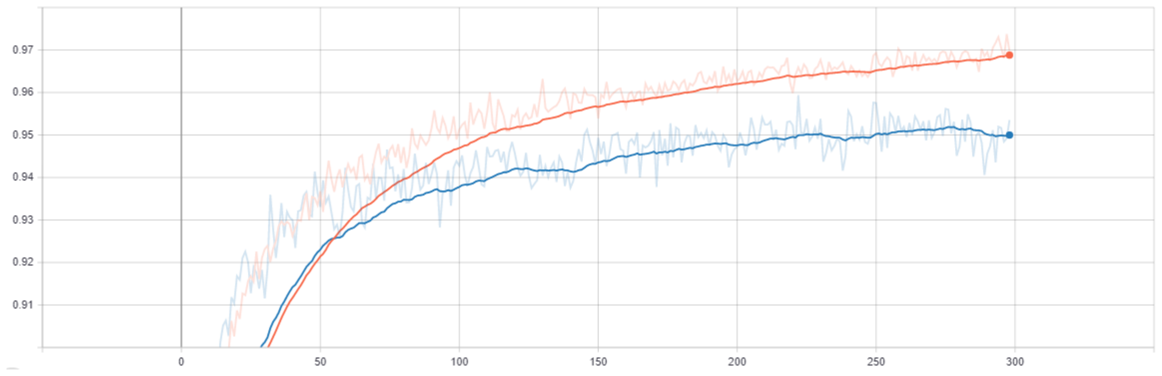


Figure 6.7: Predict Region-to-Region Network Training Accuracy (Orange = training data, Blue = evaluation data)

6.2.7.1 Results

As shown in Figure 6.7, the model reaches 95.1% accuracy over 300 training epochs.

The model is the most prone to overfitting since there are relatively few combinations of starting and target regions per level in the training data.

6.3 Predictive Engineered Exploration

Using the models developed in Section 6.2, can we improve exploration efficiency in the abstraction-guided planning algorithm? Policy simulations are relatively expensive for complex physical systems and require sequential computation. Inference in predictive models is relatively fast and can take advantage of parallelized GPU computation.

The engineered exploration described in Section 5.4.3 spends much of its time running policy simulations that fail to achieve anything useful. To reduce this wasted simulation time, Algorithm 19 modifies the Explore algorithm to skip simulation of policies that are unlikely to succeed.

Algorithm 19 PredictiveExplore$\langle R, model, predictThreshold\rangle$

```
1: $model.Regions \leftarrow model.Regions \cup R$
2: loop GetEffort($R, model$) times
3:     repeat
4:         $s \leftarrow$ SelectState($R, model$)
5:         $\pi(\theta) \leftarrow$ SelectPolicy($s, model$)
6:     until PredictSuccess($s, \pi(\theta)$) $> predictThreshold$
7:     $\tau \leftarrow$ SimulatePolicy($s, \pi(\theta)$)
8:     $s' \leftarrow$ Last($\tau$)
9:     $R' \leftarrow$ Region($s'$)
10:    if Goals($\tau$) $\neq \varnothing$ or $R' \neq R$ then
11:        $model.Trajectories(R).Add(\langle \tau, \pi(\theta) \rangle)$
12:    if $R' \neq R$ and $R' \notin model.Connections(R)$ then
13:        $model.Connections(R).Add(R')$
14: for $R' \in model.Connections(R)$ do
15:     Explore($R', model$)
```

To test the approach, the engineered exploration strategy for GF was modified to use the ExplorePredictive algorithm. PredictSuccess uses the GF predictive models developed above as follows:

- When a potential jump or roll collect is selected, the Predict Goal model predicts whether the policy will collect the specified goal.
- When a potential jump or roll transition is selected, the Predict Region model predicts whether the policy will result in a transition to the target segment.
- When an exploring jump is selected, the Predict Useful model predicts whether the exploring jump will collect any goals or transition to another segment.

6.3.1 Experiments

Each exploration strategy was evaluated on the 60 competition GF levels from 2013-2019. Results are averaged over 10 runs per level.

500 planning steps were performed for each test. Each step executes lines 5-9 of Algorithm 9, performing an Explore update with a budget of 100 simulations, followed by a Search for the best plan using the current abstract model. The best (shortest) plan time found, if any, was recorded for each step.

Since the GF levels vary considerably in difficulty and plan length, results use a relative measure of plan quality that is evaluated at each step. For each GF level plan quality Q_{level} is defined in terms of plan time T_{level} as $Q_{level} = T^*_{level}/T_{level}$ where T^*_{level} is the shortest plan time achieved in any test using any exploration strategy for that level. When a plan has not been found, T_{level} is undefined and $Q_{level} = 0$.

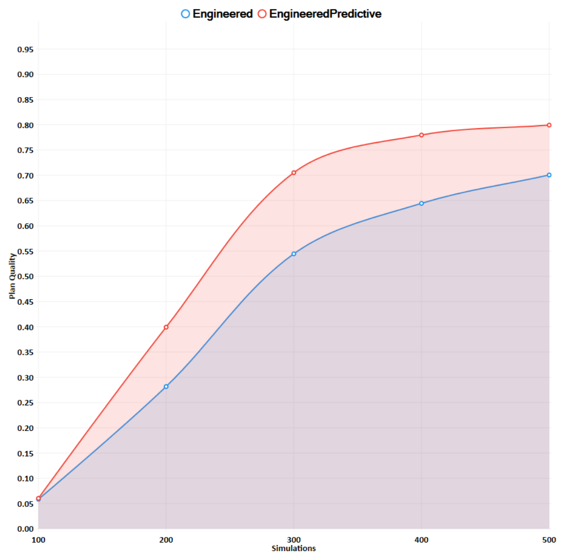
6.3.2 Results

Figure 6.8a compares plan quality as a function of simulation effort for the Engineered and Engineered Predictive exploration strategies over the first 500 simulations. The results show that on average the predictive strategy can find better quality plans with fewer simulations.

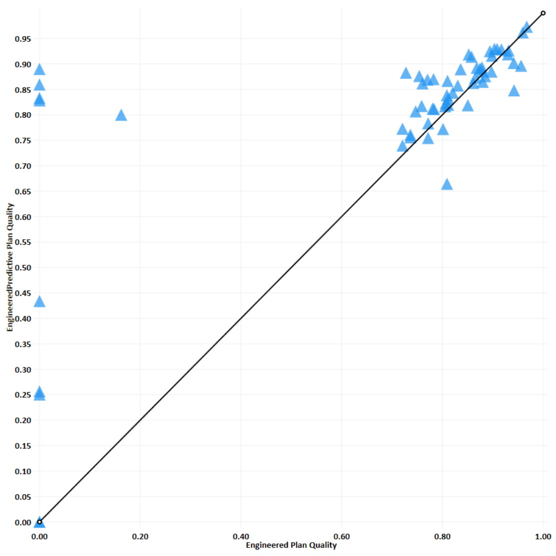
Figure 6.8b shows a scatter plot of the plan quality distribution after the first 500 simulations. Each point represents of the 60 levels tested. The x-axis is the mean plan quality for the Engineered strategy and the y-axis is the Engineered Predictive strategy. The plot shows most levels fall above the line indicating better quality using the predictive strategy.

Figure 6.8c compares plan quality over 50000 simulations. While both strategies asymptotically level off, the predictive strategy consistently performs better. Figure

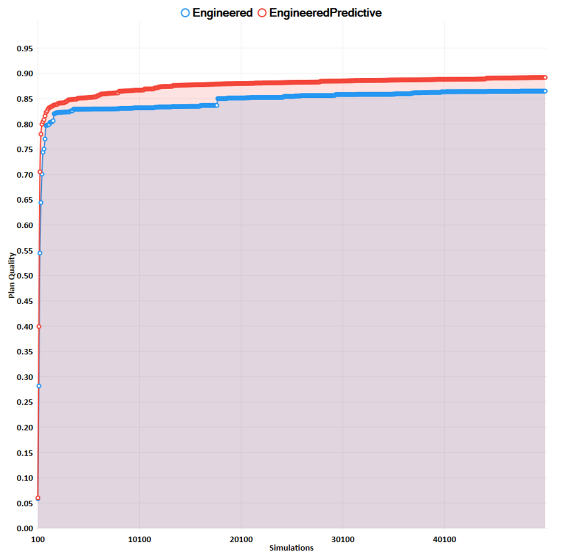
6.8d shows the 2 most difficult problems for the planner were only solvable using the predictive strategy.



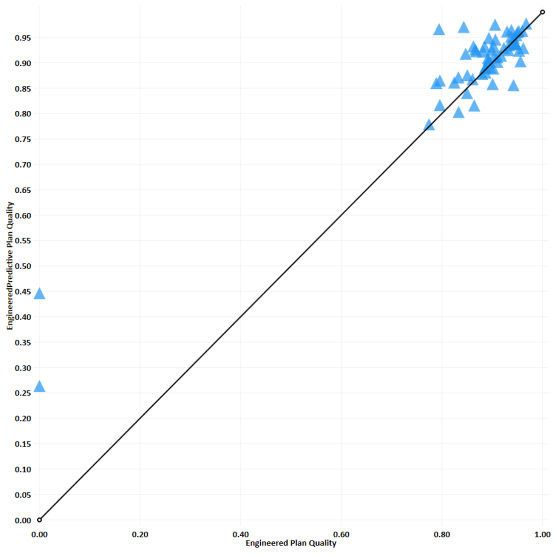
(a) Mean Plan Quality @ 500 Simulations



(b) Levels Tested @ 500 Simulations



(c) Mean Plan Quality @ 50000 Simulations



(d) Levels Tested @ 50000 Simulations

Figure 6.8: Engineered vs. Engineered Predictive Exploration

6.4 Random Exploration

The engineered exploration described in Section 5.4.3 spends most of its effort on potential transitions and goal collects that ignore obstacles, and only a small amount of effort in random exploration. As a result it performs well in levels with few obstacles but is less effective in more complex levels. It also incorporates hand-engineered expert knowledge that may not be available in other domains.

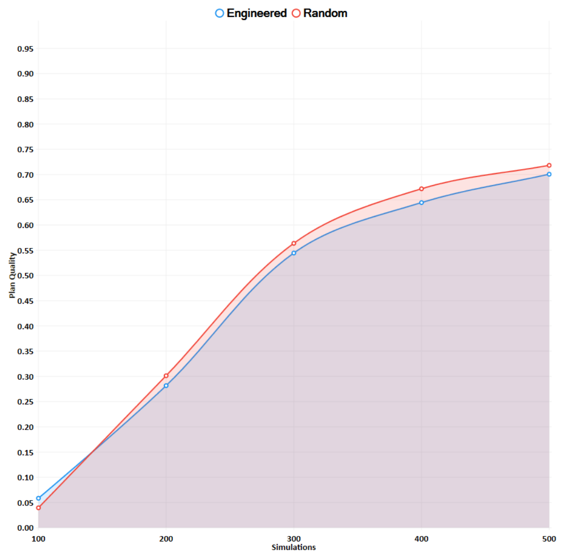
For comparison we consider an entirely random exploration strategy. For each simulation, the random strategy first selects a weighted random control policy type $\pi \in \Pi$. The weights for each policy π roughly correspond to the proportion of states in which each policy type is applicable. For GF, the policy weights were set to $\{JumpAndStop : 5, FallAndStop : 2, RollUp : 1\}$. Then state s and policy parameters θ are selected at uniform random.

6.4.1 Results

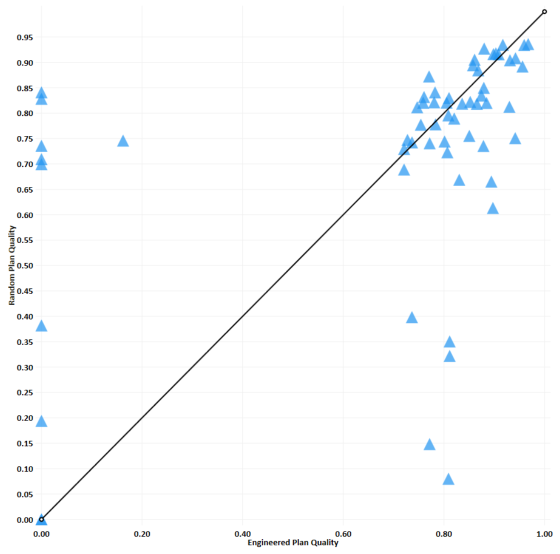
Figure 6.9a compares average plan quality as a function of simulation effort for the Engineered and Random exploration strategies for all GF levels for 500 simulations. Surprisingly, the random strategy dominates on average at all times except the initial 100 simulations. Figure 6.9b shows that the engineered strategy does in fact find better plans for close to half the levels but the random strategy finds solutions for several levels the engineered strategy cannot.

Figure 6.9c compares average plan quality over 50000 simulations. The random strategy continues to improve long after the engineered strategy levels off. Figure 6.9d shows the random strategy eventually finds better plans for almost all levels.

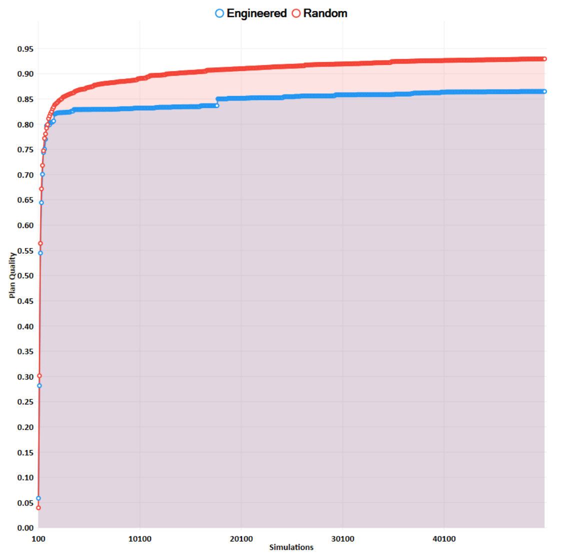
These results illustrate the potential limitations of a hand-engineered approach. By focusing on a subset of easily-predicted trajectories the agent makes quick initial progress but the strategy is slow to find more difficult transitions.



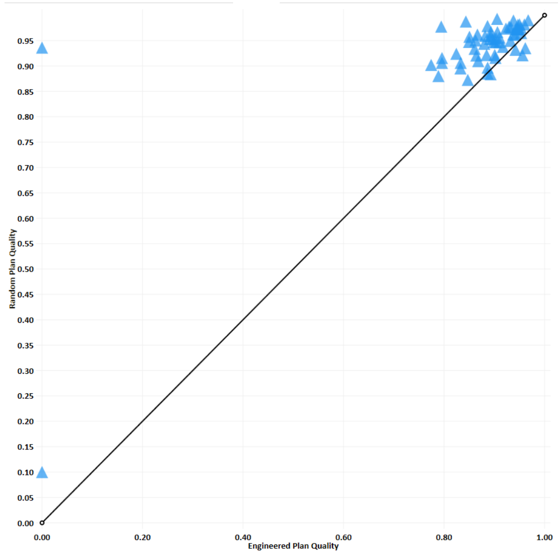
(a) Mean Plan Quality @ 500 Simulations



(b) Levels Tested @ 500 Simulations



(c) Mean Plan Quality @ 50000 Simulations



(d) Levels Tested @ 50000 Simulations

Figure 6.9: Engineered vs. Random Exploration

6.5 Random Predictive Exploration

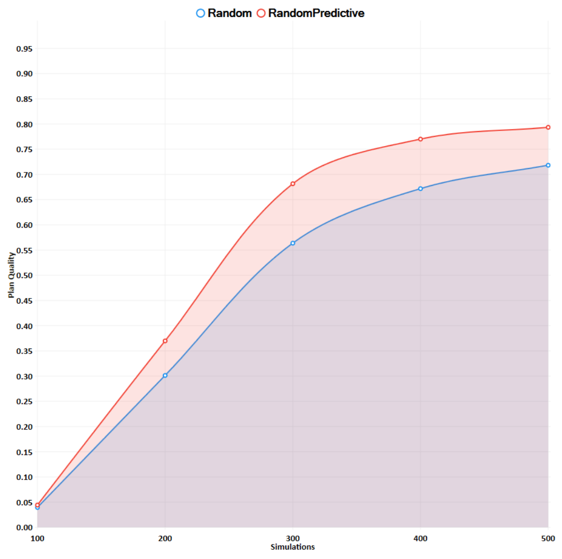
Section 6.3 showed that the use of predictive models can improve plan quality and reduce simulation time for engineered exploration sampling. Can they provide the same benefit when using entirely random sampling?

To answer this question, the random exploration strategy for GF was modified to use Algorithm 19. Each random policy sample is evaluated by Predict Useful model to predict if it will collect any goals or transition to another segment.

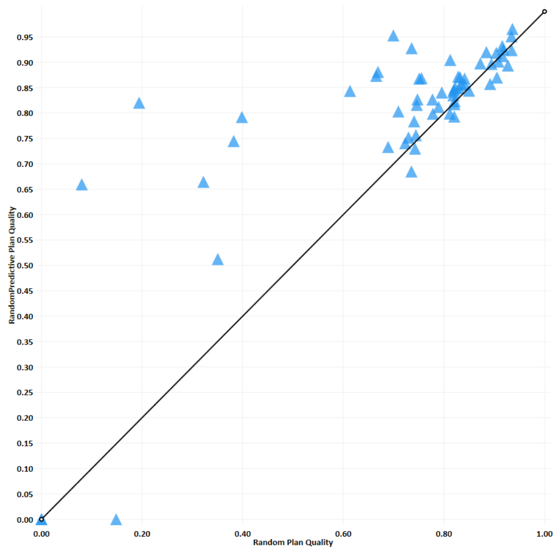
6.5.1 Results

Figure 6.10a compares average plan quality as a function of simulation effort for the Random and Random Predictive exploration strategies for all GF levels for 500 simulations. Using predictive models improves plan quality by 10-15% over the base random strategy. Figure 6.10b shows the predictive strategy outperforms the random strategy on almost all levels.

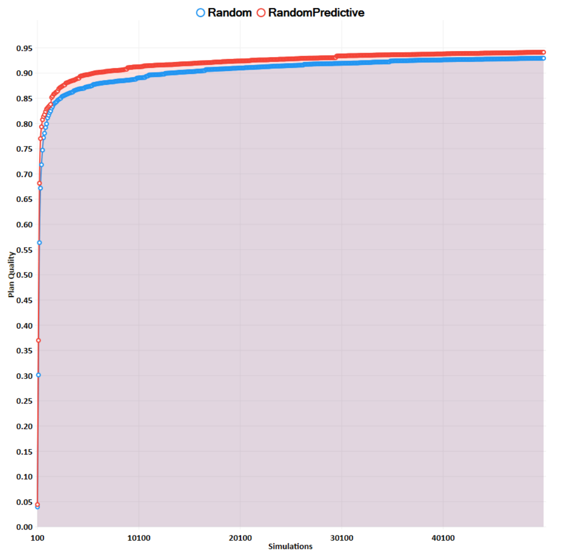
Figure 6.10c compares average plan quality over 50000 simulations. While the difference between the strategies narrows asymptotically the predictive strategy continues to result in higher plan quality. Figure 6.10d shows that both random strategies find among the highest-quality plans over time for most levels. This is as expected since these strategies will eventually sample all possible useful policies within this planning framework, given enough simulations.



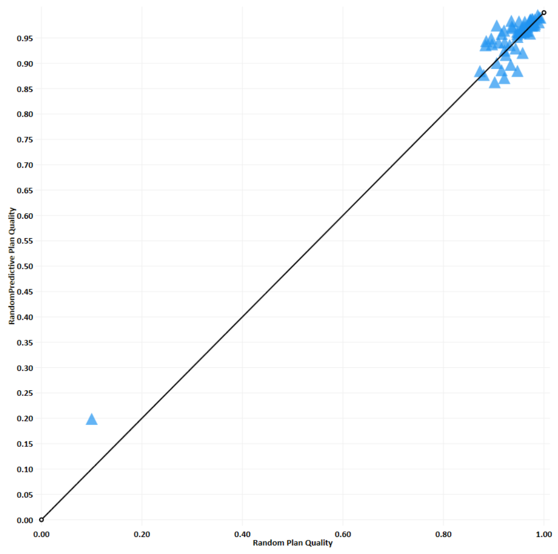
(a) Mean Plan Quality @ 500 Simulations



(b) Levels Tested @ 500 Simulations



(c) Mean Plan Quality @ 50000 Simulations



(d) Levels Tested @ 50000 Simulations

Figure 6.10: Random vs. Random Predictive Exploration

6.6 Conclusions and Further Work

This chapter has shown that predictive models can be trained to accurately estimate where policies are likely to succeed in a variety of situations. It also shows that using these predictive models, one can substantially improve the quality of plans found using a given amount of simulation resources.

The strategies used here apply the same exploration effort for every abstract state found by dividing the simulation budget equally among the states. This approach can be inefficient since it continues to allocate resources to refining the model in areas where the regions and goals have already been reached while other goals have not been achieved. Focusing attention on relevant states likely to reach remaining goals may further reduce simulation effort and exploration time to improve performance. In particular, the Predict Region-to-Goal and Predict Region-to-Region models were not utilized in the exploration strategies here. Future strategies may utilize these models to better identify abstract states to focus on.

CHAPTER 7

LEARNING ABSTRACT REPRESENTATIONS

7.1 Introduction

The GF implementation described in Chapter 5 (and indeed, all other competitive GF planners) relies on a highly domain-specific hand-engineered identification of the abstract states. One problem with such engineering is that it is typically expensive in terms of the time and expertise required. Furthermore, in some domains it may be far from obvious what a good abstraction should be.

This raises the question, is there a way to find useful abstract states automatically using general guiding principles?

This work identifies useful abstract states as regions in which a single control policy can reach any state in the region from any other state. Yet, we would like to learn control policies from scratch as well. To learn a control policy we must define the preconditions or domain of the policy, as well as the objective or goal of the policy.

This presents a chicken-and-egg problem. Abstract states define a structure for learning control policies for transitions within abstract regions and between regions. Yet control policies themselves provide the basis for identifying abstract states. Where to begin?

This chapter describes a bootstrapping approach for identifying abstract states starting with only random transition sampling and graph analysis. The identified abstract states are then used as the basis for learning a control policy for transitions within the abstract state.

7.2 Identifying Abstract States

We begin by considering how to approximate connectivity in the state space. Define approximate equivalence $s \approx s' \Rightarrow \|s - s'\| < \epsilon$ for a resolution value ϵ. Thus for a given distance threshold of ϵ, one can create a directed transition graph from a set of continuous state transition samples. The choice of Euclidean distance is arbitrary but approximates transition connectivity reasonably well for sufficiently small values of ϵ.

Our criteria for useful abstract states is that every state in an abstract region is reachable from any other state by means of a single control policy. In other words, given region R, $\forall s, s' \in R \; \exists \tau : \textsc{First}(\tau) \approx s \wedge \textsc{Last}(\tau) \approx s'$. Consequently in a complete transition graph there is a cycle for every pair of states in an abstract region.

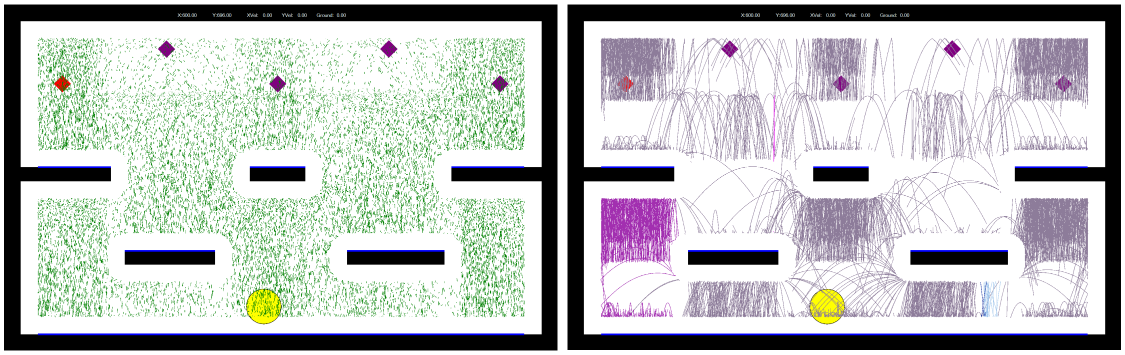
This observation suggests the following initial strategy:

- Sample transitions randomly in the state space and build a transition graph.
- Find strongly connected components in the graph.
- Regions containing dense collections of cycles may be good candidate abstract states.

7.2.1 Random Transition Sampling

To evaluate the proposed strategy, random transitions were generated for 1000 of the randomly generated GF levels described in Section 6.2.2. For each level, 1000 trajectories were generated each of length 500. At the simulation frame rate of 50Hz each trajectory represents 10 seconds of game time.

The starting location for each trajectory is selected at uniform random, discarding any locations that are in collision with obstacles. At each time step the action is



(a) Transitions (b) Strongly Connected Components

Figure 7.1: Uniform Random Transition Sampling

selected at uniform random from the set of available actions, $\mathcal{A}$. In the GF domain, $\mathcal{A} = \{NoAction, RollLeft, RollRight, Jump\}$.

Figure 7.1a visualizes the transitions for a level.

7.2.2 Transition Graph

A transition graph was constructed from the transition samples and strongly-connected components in the graph were found using Kosaraju's linear time algorithm [56]. Figure 7.1b shows the resulting connected components identified using the uniform random transitions for a distance threshold $\epsilon = 5$.

The visualization shows that the random agent spends most of its time in vertical trajectories in the air. There are virtually no rolling transitions on the platform surfaces which we have seen are crucial to engineered control in this domain.

This behavior results from the extreme differences between actions in GF. The *Jump* action immediately imparts a high y-velocity to the agent when in contact with a surface. Since the random action policy selects *Jump* action at a rate of approximately 12Hz the agent never has time to slow down and always has a high y-velocity magnitude. The resulting set of random transitions is thus strongly biased toward large y-velocities. While it achieves fairly uniform coverage of locations it does not obtain good coverage of velocities.

7.2.3 Energy-weighted Transition Sampling

In general we are interested in more uniform coverage of the state space and reducing the bias shown in Figure 7.1b. To achieve more balanced sampling of states across the full range of velocities for the agent we are motivated to consider the kinetic energy of the agent.

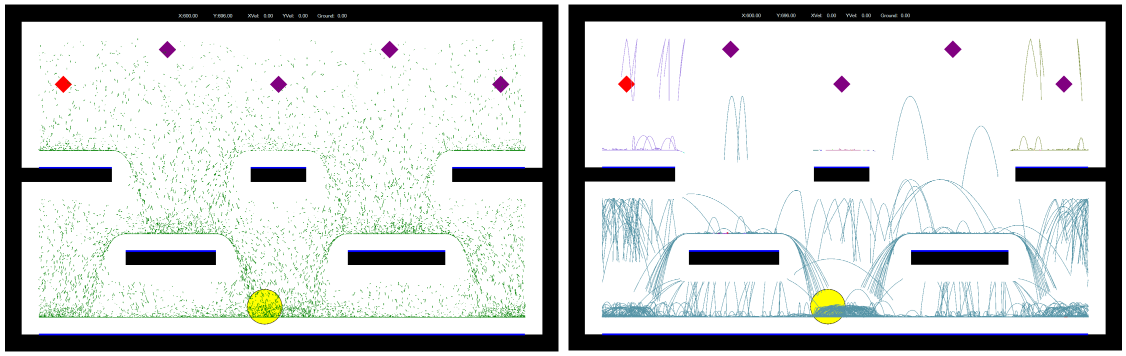
In the GF domain we observe each of the actions imparts a change in energy ΔE for time step duration δ:

- *NoAction* reduces the energy as a result of friction. Using simplistic assumptions, the force on the agent from kinetic friction is $F_k = \mu_k F_n$ where μ_k is the coefficient of kinetic friction and F_n is the normal force. Work is $force \cdot distance$ thus we can approximate $\Delta E = -\mu_k F_n |\dot{x}| \delta$.
- *RollLeft* and *RollRight* each apply a constant torque to the agent. Given torque τ and angular velocity ω, power $P = 2\pi\tau\omega$, thus $\Delta E = 2\pi\tau\omega\delta$.
- *Jump* immediately accelerates the agent to a y-velocity of $\dot{y}_{jump}$. For agent mass M the instantaneous change in kinetic energy is $\Delta E = 0.5 M \dot{y}_{jump}^2$.

These action differences suggest a modified approach for random action selection such that the probability of selecting action a varies inversely with its change in energy, i.e. $P(a) \sim 1/\Delta E_a$.

Figure 7.2 shows the effect of using this energy-weighted action selection.

The results are more balanced and the agent spends much more time on platforms.



(a) Transitions (b) Strongly Connected Components

Figure 7.2: Energy-weighted Transition Sampling

7.2.4 Shortest Cycles Continuity

By definition, every node in a strongly-connected component is reachable from any other node and thus might meet the criterion for an abstract region. Yet strongly-connected components can span large and complex regions within which it is unlikely a single control policy can reach any point. Figure 7.2b demonstrates this as a single connected component spans almost the entire lower half of the level. In fact as the density of transition sampling increases the connected components naturally grow and merge, sometimes resulting in a single component spanning the entire navigable space. Such large regions are unlikely to be useful as the basis for higher-level planning since they are too complex.

Thus, cycles among all the nodes in an abstract state is a necessary but not sufficient condition since the cycle trajectories must also be achievable by a control policy of limited complexity.

Observing the connected components as shown in Figure 7.2 reveals clusters of densely connected states connected by a sparse number of long trajectories. The shortest cycle for nodes in these long trajectories includes a large number of nodes. This suggests pruning the graph by removing these sparse long trajectories to yield more useful components, using the length of the shortest cycle as a criterion for pruning.

Consider the shortest cycle containing state s with length $\psi(s)$. Given the system dynamics and drift, $\psi(s)$ must naturally increase as the velocity of state s increases. Thus a simple threshold value for $\psi(s)$ alone is not a good criterion for pruning since it would preclude high-velocity states from abstraction.

Yet observation of the graphs also reveals that the shortest cycles for neighboring nodes within a dense cluster are relatively smooth, i.e. $|\psi(s) - \psi(s')| < \epsilon_\psi$ for edges (s, s'). Discontinuities where $|\psi(s) - \psi(s')| > \epsilon_\psi$ indicate that at least one of the nodes is not in a highly-connected cluster.

Using this insight we can analyze each connected component C in the transition graph and remove each edge (s, s') where $|\psi(s) - \psi(s')| > \epsilon_\psi$. Connected components can then be identified in the pruned graph.

The results after pruning cycle discontinuities and identifying connected components is shown in Figure 7.3. Figure 7.3a shows the location of nodes in the components. Figure 7.3b shows an alternative view that displays a vertical line with length equal to $\dot{x}$ for each node. Positive $\dot{x}$ values extend upwards and negative $\dot{x}$ values extend downwards. The velocity view reveals an interesting asymmetry as the highest $\dot{x}$ values tend to be on the right side and the most negative $\dot{x}$ values tend to be on the left. This suggests the agent is able to brake faster than it can accelerate.

The regions identified using this process correspond closely to the hand-engineered abstract states shown in Figure 7.4 which were found using geometric analysis as described in Section 5.4.1.

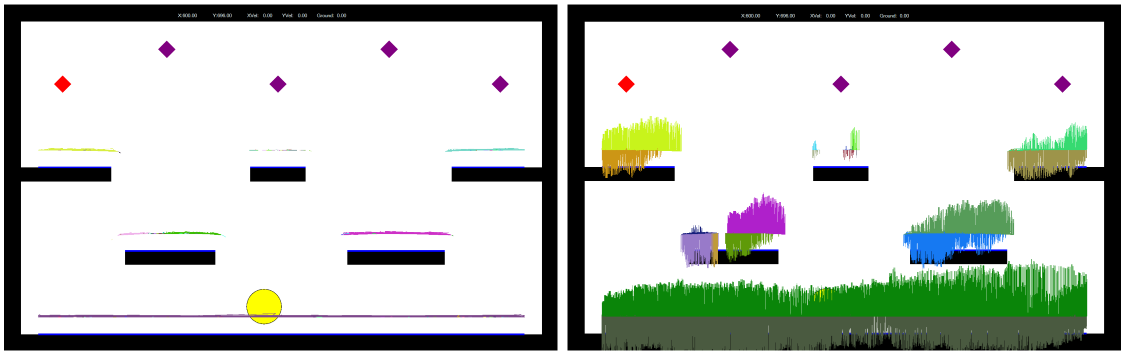

(a) Location (b) X Velocity

Figure 7.3: Abstract State Candidates

7.3 Learning a Control Policy

The GF agent in Chapter 5 uses the $RollTo(x_{target}, \dot{x}_{target})$ control policy for transitions within abstract states. The policy was learned using value iteration in only a few hours and provides high-quality control for the agent.

Yet even though the control policy is learned, it relies on a constructed learning environment with several highly-engineered domain-specific elements:

- The rolling state $s_t(x_t, \dot{x}_t, x_{target}, \dot{x}_{target})$ on which the policy operates is precisely abstracted from the full agent state $(x, \dot{x}, y, \dot{y}, \omega, O, C)$ and includes a precisely abstracted goal $(x_{target}, \dot{x}_{target})$.

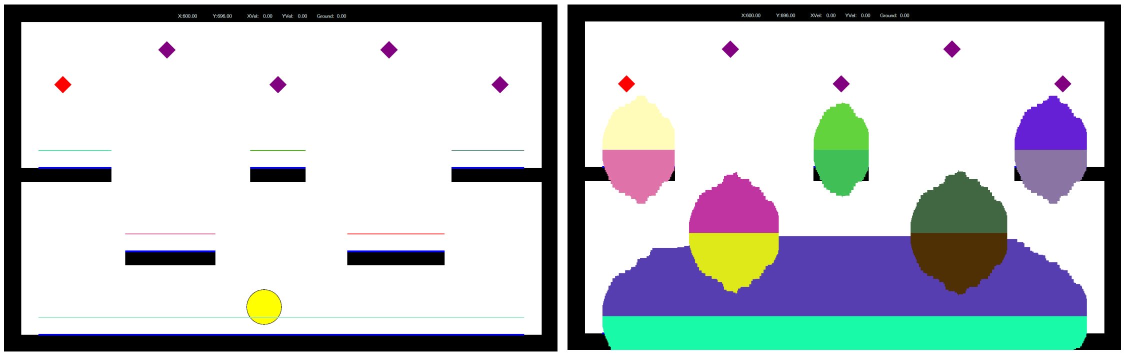

(a) Location (b) X Velocity

Figure 7.4: Engineered Abstract States

- The tabular representation using discretized states relies on expert engineering to select good resolution values for discretization.
- Learning an accurate transition model requires a implementing a hand-engineered curriculum to ensure the agent systematically obtains transition experiences across the full range of rolling states that may be encountered.

While useful, this sort of learning may be difficult or impossible to implement in other domains. Is there a more general approach that reduces the domain-specific engineering effort?

This section investigates learning a control policy from scratch for navigating within the candidate abstract regions identified in Section 7.2. The inputs to the control policy are specified using only a direct representation of the state space.

The objective is to learn a control policy $\pi(s)$ for states $s = (s_{current}, s_{target})$ where $s_{current}$ is the current actual state of the agent and s_{target} is a desired state somewhere in the current abstract region. During each training episode s_{target} remains constant while $s_{current}$ updates at each time step according to the system dynamics.

The learning agent for the control policy was implemented using a Deep-Q Network (DQN) architecture described below.

7.3.1 Agent Architecture

The Deep-Q Network (DQN) architecture [40] is basis of much recent work utilizing deep convolutional neural networks with reinforcement learning. It has proven highly effective for a wide range of challenging domains such as Atari video games using no significant prior knowledge of the problem domain, and in some cases surpassing human expert performance.

DQN utilizes a variant of Q-learning [57], an off-policy model-free method that directly learns an action-value function $Q(s, a; \theta)$ that approximates the value of taking action a in state s using a convolutional neural network with parameters θ. This type of network is called a Q-network.

Future rewards are discounted by a factor of γ per time step so that the *total return* at time t is $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$ where r_i is the reward at time i. For these experiments $\gamma = 0.99$. Any optimal Q function Q^* satisfies the Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s' \sim p(\cdot|s,a)}[r(s,a) + \gamma \max_{a' \in A} Q^*(s',a')] \tag{7.1}$$

A *greedy action* in state s is one that selects the highest expected return, or $a^* = \operatorname{argmax}_{a \in A} Q^*(s,a)$.

To increase stability and increase the likelihood of convergence of the network, the DQN incorporates 2 copies of the Q-network, the primary network Q with parameters θ and an identical *target network* with parameters $\bar{\theta}$. θ is updated during each batch update and is periodically copied to $\bar{\theta}$.

At each time step t the agent uses ϵ-greedy action selection such that the agent selects a random action with probability ϵ, otherwise it selects the greedy action with respect to Q_θ. Each step (s_t, a_t, r_t, s_{t+1}) is recorded in an *experience replay* buffer containing the last 2.5×10^6 experiences.

The network parameters θ are updated iteratively using batch stochastic gradient descent with the Adam optimizer, but $\bar{\theta}$ is held constant during updates. Gradient descent minimizes the following loss function:

$$L = (r_t + \gamma \max_{a' \in A} Q_{\bar{\theta}}(s_{t+1}, a') - Q_\theta(s_t, a_t))^2 \tag{7.2}$$

Target update values $r_t + \gamma \max_{a' \in A} Q_{\bar{\theta}}(s_{t+1}, a')$ are clipped to the range [0..1000].

Each training update selects a batch of experiences (s_t, a_t, r_t, s_{t+1}) at uniform random from the replay buffer. The large replay buffer improves efficiency by reusing experiences and is crucial for reducing the correlation that arises from consecutive experiences.

The reward function used is:

$$r(s) \leftarrow \begin{cases} 1000 & \text{if } s \text{ is a terminal state} \\ 0 & \text{otherwise} \end{cases} \tag{7.3}$$

A state $s = (s_{current}, s_{target})$ is a terminal state if $\|s_{current} - s_{target}\| < \epsilon_{terminal}$.

Since rewards are sparse and received only when successfully reaching a target state, the agent may go for long periods without receiving positive reward. Hindsight experience replay [3] can speed up goal-directed training in these situations by adding artificial experiences to the replay buffer that simulate pursuing goals selected from the current episode. These experiences represent what could have happened for the actual state/action transitions if the agent had a different goal closer to the current state. For each actual experience (s_t, a_t, r_t, s_{t+1}) recorded, the agent may record k additional synthetic experiences $(\sigma, a_t, r(\sigma'), \sigma')$ where $\sigma = (s_t.current, g)$ and $\sigma' = (s_{t+1}.current, g)$ with goal state $g = s.current$ for a random state s that occurs after time t in the current episode. The hindsight parameter k used was 1, resulting in an equal proportion of real and synthetic hindsight experience.

The network structure is shown in Figure 7.5. The 7 input channels assign the workspace pixel values as follows:

- Obstacles: 1 if occupied by an obstacle, else 0
- Agent Location: 1 if occupied by the agent, else 0
- Agent X Velocity: $\dot{x}$ if occupied by the agent, else 0
- Agent Y Velocity: $\dot{y}$ if occupied by the agent, else 0
- Goal Location: 1 if occupied by the agent target location, else 0

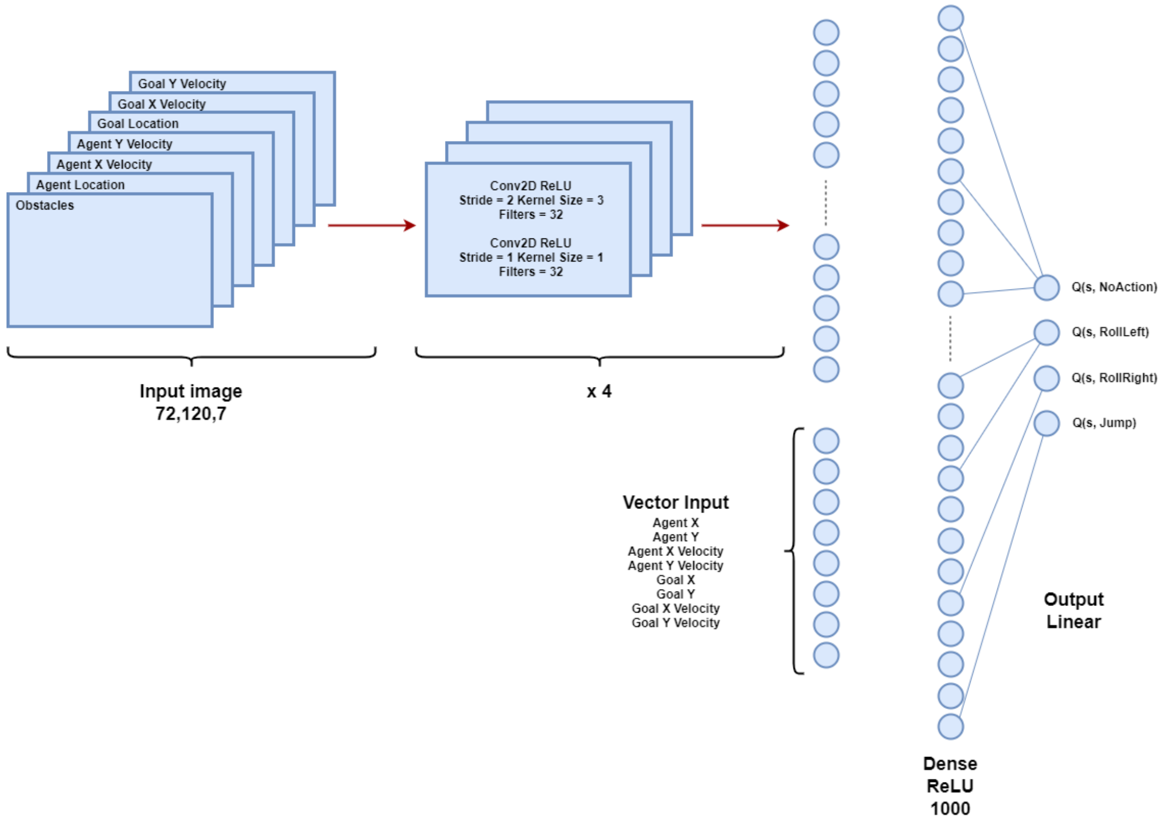


Figure 7.5: Deep Q Network Architecture

- Goal X Velocity: $\dot{x}_{target}$ if occupied by the agent target location, else 0
- Goal Y Velocity: $\dot{y}_{target}$ if occupied by the agent target location, else 0

Each episode ends when the agent reaches the goal, or after 250 time steps.

7.3.2 Results

Figure 7.6 shows the average reward per episode for the best agent tested over the course of approximately 1.8×10^5 batch updates. The average reward approaches 600 which represents about 60% episode success rate since the reward is 1000 for reaching the goal and 0 otherwise.

Figure 7.7 shows the average change in distance to the goal for the same agent where Δ-distance$= \|s_{final} - s_{target}\| - \|s_{initial} - s_{target}\|$. Negative values for Δ-distance

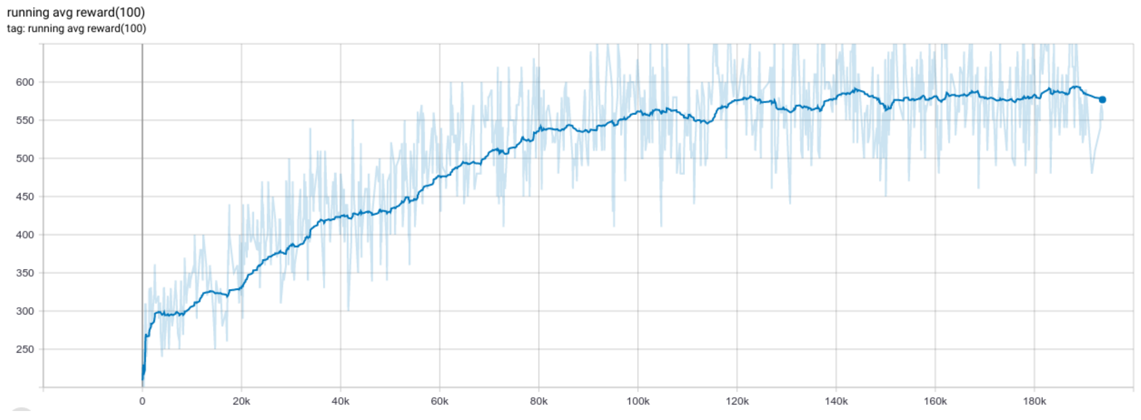


Figure 7.6: Deep Q Agent Average Reward

indicate the agent is moving closer to the goal in each episode and shows progress in approaching goals even if the agent does not reach the goal precisely. The results show the agent is clearly making progress at approaching goal states even though it does not reach them 40% of the time.

Figure 7.8 compares the DQN agent (red) to a random agent (blue). The histogram shows the Euclidean distance between the agent and goal at the end of each episode for 2500 episodes. The DQN agent is over twice as likely to end close to the goal as a random agent and is much less likely to end far from the goal.

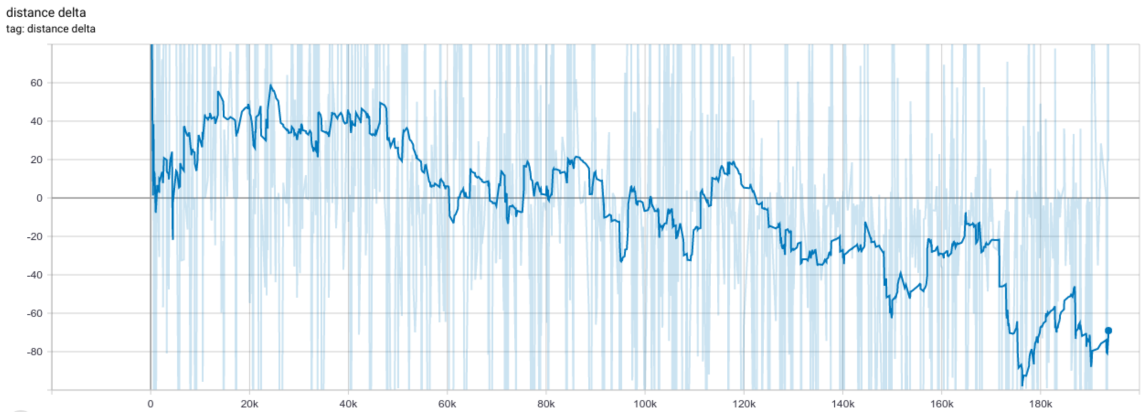


Figure 7.7: Deep Q Agent Δ-distance

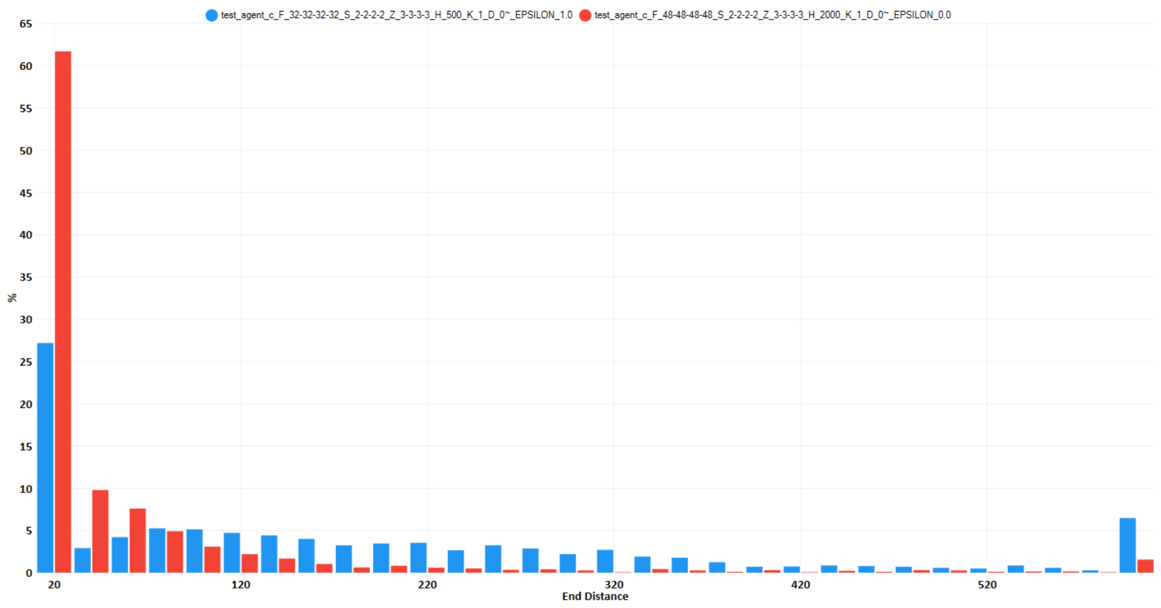


Figure 7.8: Deep Q Agent End Distance vs. Random Agent

A small number of variations of the convolutional network architecture were tested:

- Model A : Conv2D + ReLU
- Model B : Conv2D + ReLU + batch normalization
- Model C : Conv2D + ReLU + Conv2D 1x1 + ReLU
- Model D : Conv2D + ReLU + Conv2D 1x1 + ReLU + batch normalization
- Model E : Vector input only, no convolution or image input

The results are shown in Figure 7.9. For each agent the number of convolutional filters is shown and the number of hidden nodes. All agents used hindsight experience replay with $k = 1$ except for one.

The performance of the best agent is still far from optimal and the results suggest some general guidelines for further improvement.

- Hindsight experience replay greatly improves learning for these agents. The only agent tested without hindsight experience had the worst performance of

Agent	Solved %	Steps	Min Distance	End Distance	Delta Distance
Random	26.8	187.2	86.1	186.8	77.1
A 32x32x32x32 H 500	**58.8**	**137.7**	56.1	84.0	-29.7
B 32x32x32x32 H 500	28.3	187.0	75.1	157.8	51.1
C 32x32x32x32x32x32 H 500	56.7	142.8	51.8	83.2	-21.9
C 32x32x32x32x32 H 500	57.2	143.5	56.1	91.1	-25.4
C 32x32x32x32 H 500 No Hindsight	32.4	180.3	73.9	178.6	70.0
C 32x32x32x32 H 500	50.0	154.4	38.4	66.5	-40.3
C 48x48x48x48 H 1000	55.0	150.5	40.7	68.3	-40.5
C 48x48x48x48 H 2000	57.7	143.4	**34.3**	**55.3**	**-48.7**
D 32x32x32x32 H 500	29.6	183.1	73.9	145.6	35.8
E H 1000x1000x1000x1000	46.8	152.5	41.8	75.5	-23.6
E H 1000x1000x1000	41.8	160.3	66.0	121.1	8.0
E H 1000x1000	40.9	159.9	75.3	171.0	59.9
E H 2500x2500	40.6	161.8	65.0	160.1	58.9

Figure 7.9: Deep Q Agent Mean Results 2500 Episodes

all agents, with an average End Distance from the goal only slightly less than the random agent.

- Regularization in Q-networks can adversely affect performance. Models B and D used batch normalization in the convolution layers and had among the worst results of all agents tested, only marginally better than the random agent.
- Convolution yields better results in comparison to direct vector input. Model E did not use convolution and had poor results overall.
- More complex models may be required. Model C added an extra 1x1 convolution for each of the convolutional layers and had achieved consistently better results than all other models. Furthermore the best agent tested had the largest number of filters and hidden nodes of all the model C agents.

7.4 Conclusions and Further Work

This chapter has demonstrated a process using random sampling and graph analysis that yields candidate abstract states in GF that closely resemble the hand-engineered abstractions created by human experts in competitive planners. In addition, a deep-Q learning agent with minimal domain-specific engineering demonstrated ability to navigate within the regions identified.

This work is a first step towards planning using entirely learned abstract states and control policies.

The work here has identified abstract states only for the GF domain. A priority for future work is applying and evaluating the ability of the method to generalize for other diverse domains.

Additional work may explore iteratively expanding and refining the initial abstract states with focused transition sampling in the neighborhood of each state. An area of interest is whether using we can use the learned control policy itself to better define and expand the boundaries of the abstract regions.

The bootstrap process for identifying abstract states presented here is relatively time-consuming and must be performed separately for each environment with a unique obstacle configuration. Future work will investigate learning models to identify abstract states quickly and reduce or eliminate the sampling and graph analysis required.

The objective of the agent control policy learned here is to reach other states within the same abstract region. Future work will consider learning 1 or more additional control policies that can transition from one region to another, which is a requirement for abstract planning.

The Q-learning method used by DQN uses only 1-step backups for Q updates, while the high frame rate means that rewards are often more than 100 steps distant.

Together with the large state space, this results in very slow propagation of Q-values for the agent policy. *Advantage learning* [25] is similar to Q-learning except that it uses advantages rather than Q-values. Advantage learning can sometimes learn orders of magnitude faster than Q-learning when using a function approximator, particularly when the state changes for each time step are small. Further work will evaluate whether advantage learning may be more efficient for this domain.

Further work will also refine and improve performance of the learning agent for the GF control policy and explore potentially more efficient learning based on SARSA or other on-policy methods that propagate rewards to distant states faster.

CHAPTER 8

CONCLUSIONS

8.1 Summary

This dissertation has investigated novel applications of machine learning to improve the efficiency and quality of planning on both symbolic and continuous problem domains. This chapter reviews the key contributions presented and suggests directions for future work.

8.1.1 Dynamic Heuristic Selection

A novel approach for planning based on heuristic search was presented, in which the planner decides how to allocate computational effort based on estimates of search progress. By observing heuristic search progress over a period of time, the planner extracts features of the search dynamics that are useful for learning and online decision-making during planning. Given a set of training problems drawn from a target problem distribution, and given a base set of domain-independent heuristic planners, a predictive model is trained to estimate heuristic progress on problems in the domain. After training, these estimates are then used to dynamically select the most promising heuristic during the planning process. This is the first work that uses a learned model of the heuristic search dynamics to dynamically select a heuristic or planner during planning. Experimental results on planning benchmark problems show that dynamic heuristic selection using the learned model can solve more problems than static approaches that select base planners before planning starts.

Future work may explore further properties and applications of heuristic search dynamics in planning. DH1 is a portfolio planner and the base heuristic planners are entirely independent of one another. Search dynamics are used only for switching between base planners. Still to be investigated is the use of search dynamics within a base planner to select between multiple heuristics with shared open lists instead of using fixed strategies such as alternation. This may improve performance if some heuristics are more informative than others depending on the region of the search space.

In DH1 the regression function is used solely for selecting a base heuristic planner to execute. Another question is whether the regression estimator can be used to better determine when a planner should utilize random exploration or a deep local search in order to escape a local minima in conjunction with algorithms such as [65].

The window feature extraction function used here is hand-coded and fixed. An open question is whether more effective features can be learned directly from the data. The predictive models used in this work utilized support vector machines. Deep neural networks that can learn useful features may be a better approach.

8.1.2 Balanced k-Nearest Neighbors

To address the problem of local neighborhood bias described in Section 2.8, this work introduced two new algorithms for k-nearest neighbor (kNN) regression and classification. Axis-balanced kNN and Box kNN compensate for non-uniform training sample distribution by adjusting the weights of the k-nearest neighbors to balance the influence of samples from opposing regions of space. Axis-balanced kNN adjusts the weights of the k-nearest neighbors to approximate a balanced distribution along each feature axis. Box kNN, in contrast, adjusts the weights of the k-nearest neighbors to include only the nearest neighbors in each feature axis direction. Neither method requires additional parameters or tuning beyond that required by kNN.

Experiments using synthetic and real-world data demonstrate these methods can improve accuracy in comparison to kNN. Axis-balanced kNN performs better when training data is more dense and there is little to moderate noise. Box-kNN tends to outperform the other approaches when training data is less dense and there is little noise. When there is a high level of noise, or the intrinsic dimensionality of the data is 10 or higher, standard kNN is likely the best choice.

The balancing approach presented here is performed solely along the feature axes of the original sample data. To further improve performance, balancing in other directions may be more effective, such as using whitened data. Another potential improvement is to first globally transform the sample data using distance metric learning methods for classification problems.

8.1.3 Abstraction-Guided Planning

This work presented an abstraction-guided kinodynamic planning algorithm for multi-goal problems using control policies.

Rather than basing abstract states on a simple arbitrary geometric decomposition of space, the approach forms a useful abstraction of the state space in relation to the control policies. Abstract states correspond to regions in the state space such that any state is reachable from any other state in the region using one of the control policies.

The planner explores reachable space using the control policies and builds a directed graph of abstract states. Connectivity between abstract states is established by applying control policies to states sampled from a region and recording transitions to other abstract state regions or goal regions. The resulting graph is used to search for an abstract high-level plan that is dynamically feasible and achieves all of the goals.

The algorithm was implemented and evaluated in a real-time planning agent for the Geometry Friends competition held at the 2019 IEEE Conference on Games. Competition results demonstrate planner performance is comparable to a more highly-engineered domain-specific planner, and significantly outperforms a more general sampling-based planner.

Further work will extend the GF implementation of AGAgent to support planning for the Rectangle and Cooperative games. These variations of the game competition change the action and state spaces and can provide additional insight into the requirements for adapting the algorithm to new domains.

More broadly, an open question to be answered is how well the AGAgent planner can generalize to other challenging domains that require dynamic interaction with obstacles, such as foot step planning and wall climbing.

8.1.4 Predictive Exploration

The abstraction-guided GF competition agent uses a highly-engineered exploration strategy tailored to the problem domain in order to sample policies for simulation.

To improve upon the engineered strategy, this work learned accurate predictive models for estimating how useful a control policy is likely to be in a range of situations. These predictive models were then incorporated into the exploration process to avoid simulating policies that have little chance of success.

Results show that augmenting the engineered exploration with the predictive strategy results in improved plan quality compared to engineered exploration alone, given the same amount of simulation resources.

In addition, pure random exploration was also augmented with the predictive strategy. Again, predictive exploration showed consistent improvement over the baseline exploration strategy.

The strategies evaluated here all apply the same exploration effort for every abstract state found by dividing the simulation budget equally among the states. This approach can be inefficient since it continues to allocate resources to refining the model in areas where the regions and goals have already been reached while other goals have not been achieved. Focusing attention on relevant states likely to reach remaining goals may further reduce simulation effort and exploration time to improve performance.

8.1.5 Learning Abstract States

This work has proposed a novel approach using random sampling and graph analysis for learning abstract states, with the goal of identifying useful state abstractions using only a general set of guiding principles.

The approach is shown to yield candidate abstract states in GF that closely resemble the hand-engineered abstractions used by competitive planners. In addition, a deep-Q learning agent with minimal engineering demonstrated the ability to navigate within the regions identified.

This work represents a first step towards hybrid symbolic/continuous planning using entirely learned abstract states and control policies.

The work here has identified abstract states only for the GF domain. A priority for future work is applying and evaluating the method for other diverse domains.

Additional work may explore iteratively expanding and refining the initial abstract states with focused transition sampling in the neighborhood of each state. An area of interest is whether we can use the learned control policy itself to better define and expand the boundaries of the abstract regions.

The bootstrap process for identifying abstract states presented here is relatively time-consuming and must be performed separately for each environment with a unique obstacle configuration. Future work will investigate learning models to identify abstract states quickly and reduce or eliminate the sampling and graph analysis required.

The objective of the agent control policy learned here is to reach other states within the same abstract region. Future work will consider learning 1 or more additional control policies that can transition from one region to another, which is a requirement for abstract planning.

The Q-learning method used by DQN uses only 1-step backups for Q updates, while the high frame rate means that rewards are often many steps distant. Together with the large state space, this results in very slow propagation of Q-values for the agent control policy. *Advantage learning* [25] is similar to Q-learning except that it is guided by relative advantage of actions rather than the Q-values directly. Advantage learning can sometimes learn orders of magnitude faster than Q-learning when using function approximation, particularly when the state changes for each time step are small. Further work will evaluate whether advantage learning may be useful for this problem.

Further work will seek to refine and improve performance of the learning agent for the GF control policy and explore potentially more efficient learning based on SARSA or other on-policy methods, as well as alternative neural network architectures.

REFERENCES

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mane, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viegas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. Tech. rep. (2015), `http://tensorflow.org/`

[2] Allen, R.E., Clark, A.A., Starek, J.A., Pavone, M.: A machine learning approach for real-time reachability analysis. EEE/RSJ International Conference on Intelligent Robots and Systems pp. 2202–2208 (2014)

[3] Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., McGrew, B., Tobin, J., Abbeel, P., Zaremba, W.: Hindsight Experience Replay. In: Advances in Neural Information Processing Systems. pp. 5048–5058 (2017)

[4] Barley, M., Franco, S., Riddle, P.: Overcoming the Utility Problem in Heuristic Generation: Why Time Matters. In: Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling. pp. 38–46 (2014)

[5] Bharatheesha, M., Caarls, W., Wolfslag, W.J., Wisse, M.: Distance metric approximation for state-space RRTs using supervised learning. In: IEEE International Conference on Intelligent Robots and Systems. pp. 252–257 (2014)

[6] Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence 129(1), 5–33 (2001)

[7] Browning, B., Bruce, J., Bowling, M., Veloso, M.: STP: Skills, tactics, and plays for multi-robot control in adversarial environments. Proceedings of the Institution of Mechanical Engineers. Part I: Journal of Systems and Control Engineering 219(2), 33–52 (2005)

[8] Burns, B., Brock, O.: Sampling-based motion planning using predictive models. IEEE International Conference on Robotics and Automation pp. 3120 – 3125 (2005)

[9] Cenamor, I., de la Rosa, T., Fernandez, F.: IBACOP and IBACOP2 Planner. In: IPC 2014 planner abstracts. pp. 35–38 (2014)

[10] Cheng, P., Lavalle, S.M.: Reducing metric sensitivity in randomized trajectory design. Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180) 1, 43–48 (2001)

[11] Chollet, F.: Keras: The python deep learning library. Astrophysics Source Code Library (2018), `https://keras.io`

[12] Cook, B., Huber, M.: Dynamic heuristic planner selection. In: 2016 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2016 - Conference Proceedings (2016)

[13] Cook, B., Huber, M.: Balanced k-Nearest Neighbors. In: FLAIRS-32. pp. 112–115 (2019)

[14] Cross, S.E., Walker, E.: Applying Knowledge-based Planning and Scheduling to Crisis Action Planning. Intelligent Scheduling pp. 711–29 (1994)

[15] Dheeru, D., Karra Taniskidou, E.: UCI Machine Learning Repository (2017), `http://archive.ics.uci.edu/ml`

[16] Diankov, R., Kuffner, J.: Randomized statistical path planning. IEEE International Conference on Intelligent Robots and Systems pp. 1–6 (2007)

[17] Domshlak, C., Karpas, E., Markovitch, S.: To Max or Not to Max: Online Learning for Speeding Up Optimal Planning. In: AAAI (2010)

[18] Dudani, S.A.: The Distance-Weighted k-Nearest-Neighbor Rule. IEEE Trans. on Systems, Man, and Cybernetics 6(4), 325–327 (apr 1976)

[19] Edelkamp, S., Plaku, E.: Multi-goal motion planning with physics-based game engines. In: IEEE Conference on Computatonal Intelligence and Games, CIG (2014)

[20] Fikes, R.E., Nilsson, N.J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2(3-4), 189–208 (1971)

[21] Friedman, J., Hastie, T., Tibshirani, R.: The elements of statistical learning, vol. 1. Springer series in statistics New York, NY, USA (2001)

[22] Garcia-Pedrajas, N., Romero Del Castillo, J.A., Cerruela-Garcia, G.: A Proposal for Local k Values for k-Nearest Neighbor Rule. IEEE Trans. on Neural Networks and Learning Systems 28(2), 470–475 (2017)

[23] Gerevini, A., Saetti, A., Vallati, M.: PbP2: Automatic Configuration of a Portfolio-based Multi-Planner. In: The 2011 International Planning Competition (2011)

[24] Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory and Practice. Elsevier (2004)

[25] Harmon, M.E., Baird III, L.C.: Multi-Agent Residual Advantage Learning with General Function Approximation. Tech. rep., Wright Laboratory, Wright-Patterson Air Force Base, OH (1996)

[26] Hastie, T., Tibshirani, R.: Discriminant adaptive nearest neighbor classification. IEEE Trans. on Pattern Analysis and Machine Intelligence 18(6), 607–616 (1996)

[27] Helmert, M.: The fast downward planning system. Journal of Artificial Intelligence Research 26, 191–246 (2006)

[28] Helmert, M., Geffner, H.: Unifying the Causal Graph and Additive Heuristics. In: International Conference on Automated Planning and Scheduling. pp. 140–147 (2008)

[29] Helmert, M., Röger, G., Seipp, J., Karpas, E., Hoffmann, J., Keyder, E., Nissim, R., Richter, S., Westphal, M.: Fast downward stone soup. In: The 2011 International Planning Competition. pp. 38–45 (2011)

[30] Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302 (2001)

[31] Kalisiak, M., Van De Panne, M.: RRT-blossom: RRT with a local flood-fill behavior. Proceedings - IEEE International Conference on Robotics and Automation 2006, 1237–1242 (2006)

[32] Kalisiak, M., Van De Panne, M.: Faster motion planning using learned local viability models. Proceedings - IEEE International Conference on Robotics and Automation pp. 2700–2705 (2007)

[33] Karaman, S., Frazzoli, E.: Sampling-based Algorithms for Optimal Motion Planning 30(7), 846–894 (2011)

[34] Kavraki, L., Svestka, P., Overmars, M.: Probabilistic roadmaps for path planning in high-dimensional configuration spaces. Tech. rep. (1994)

[35] Lamiraux, F., Ferre, E., Vallee, E.: Kinodynamic motion planning: connecting exploration trees using trajectory optimization methods. Proc. IEEE International Conference on Robotics and Automation ICRA '04 4, 3987—-3992 Vol.4 (2004)

[36] Lavalle, S.M.: Rapidly-Exploring Random Trees: A New Tool for Path Planning. Tech. rep. (1998)

[37] LaValle, S.M.: Planning Algorithms. Cambridge University Press (2006)

[38] Lecun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature 521(7553), 436–444 (2015)

[39] McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D.: PDDL - The Planning Domain Definition Language. Annals of Physics 54, 26 (1998)

[40] Mnih, V., Silver, D., Riedmiller, M.: Playing Atari with Deep Reinforcement Learning. Tech. rep. (2013)

[41] Mu, Y., Ding, W., Tao, D.: Local discriminative distance metrics ensemble learning. Pattern Recognition 46(8), 2337–2349 (2013)

[42] Nechushtan, O., Raveh, B., Halperin, D.: Sampling-diagram automata: A tool for analyzing path quality in tree planners. Springer Tracts in Advanced Robotics 68(STAR), 285–301 (2010)

[43] Oonishi, H., Iima, H.: Improving generalization ability in a puzzle game using reinforcement learning. In: 2017 IEEE Conference on Computational Intelligence and Games. pp. 232–239 (2017)

[44] Palmieri, L., Arras, K.O.: Distance Metric Learning for RRT-Based Motion Planning for Wheeled Mobile Robots. International Conference on Robotics and Automation pp. 637–643 (2015)

[45] Pan, Z., Wang, Y., Ku, W.: A new general nearest neighbor classification based on the mutual neighborhood information. Knowledge-Based Systems 121, 142–152 (2017)

[46] Perez, D., Powley, E.J., Whitehouse, D., Rohlfshagen, P., Samothrakis, S., Cowling, P.I., Lucas, S.M.: Solving the physical traveling salesman problem: Tree search and macro actions. IEEE Transactions on Computational Intelligence and AI in Games 6(1), 31–45 (2014)

[47] Perez, D., Robles, D., Rohlfshagen, P.: The Physical Travelling Salesman Problem Competition Website pp. 10–15 (2012), `http://ptsp-game.net/`

[48] Phillips, M., Mellon, C., Narayanan, V., Aine, S., Mellon, C., Technology, I., Likhachev, M., Mellon, C.: Efficient Search with an Ensemble of Heuristics. In: Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (2015)

[49] Plaku, E., Kavraki, L.E., Vardi, M.Y.: Motion Planning With Dynamics by a Synergistic Combination of Layers of Planning pp. 1–14 (2010)

[50] Powley, E.J., Whitehouse, D., Cowling, P.I.: Monte Carlo tree search with macro-actions and heuristic route planning for the multiobjective physical travelling salesman problem. In: 2013 IEEE Conference on Computational Intelligence in Games (CIG 2013). pp. 1–8 (2013)

[51] Prada, R., Lopes, P., Catarino, J., Quitério, J., Melo, F.S.: The Geometry Friends Game AI Competition. IEEE Conference on Computational Intelligence and Games (CIG'15) pp. 431–438 (2015)

[52] Richter, S., Westphal, M.: The LAMA planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research 39(1), 127–177 (2010)

[53] Richter, S., Westphal, M., Helmert, M.: LAMA 2008 and 2011. In: The 2011 International Planning Competition. pp. 117–124 (2011)

[54] Röger, G., Helmert, M.: The more, the merrier: Combining heuristic estimators for satisficing planning. In: International Conference on Automated Planning and Scheduling. pp. 246–249 (2010)

[55] Salta, A., Prada, R., Melo, F.S.: A new approach for Geometry Friends' RRT Agents. In: Geometry Friends AI Competition @ EPIA 2017 (2017)

[56] Sharir, M.: A strong-connectivity algorithm and its applications in data flow analysis. Computers and Mathematics with Applications 7(1), 67–72 (1981)

[57] Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)

[58] Titterington, D.M.: A Re-Examination of the Distance-Weighted k-Nearest Neighbor Classification Rule. IEEE Trans. on Systems, Man, and Cybernetics 17(4), 689–696 (1987)

[59] Wang, J., Neskovic, P., Cooper, L.N.: Improving nearest neighbor rule with a simple adaptive distance measure. Pattern Recognition Letters 28(2), 207–213 (2007)

[60] Wang, J., Woznica, A., Kalousis, A.: Parametric Local Metric Learning for Nearest Neighbor Classification pp. 1–9 (2012)

[61] Weinberger, K.Q., Saul, L.K.: Distance Metric Learning for Large Margin Nearest Neighbor Classification. Journal of Machine Learning Research 10, 207–244 (2009)

[62] Wolfslag, W., Bharatheesha, M., Moerland, T., Wisse, M.: RRT-CoLearn: towards kinodynamic planning without numerical trajectory optimization. IEEE Robotics and Automation Letters 3(3), 1655–1662 (2018)

[63] Wolpert, D.H., Macready, W.G.: No free lunch theorems for search. Tech. rep., SFI-TR-05-010, The Santa Fe Institute (1995)

[64] Wu, X., Kumar, V., Ross, Q.J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G.J., Ng, A., Liu, B., Yu, P.S., Zhou, Z.H., Steinbach, M., Hand, D.J., Steinberg, D.: Top 10 Algorithms in Data Mining. Knowledge and Information Systems 14(1), 1–37 (2008)

[65] Xie, F., Müller, M., Holte, R.: Jasper : the Art of Exploration in Greedy Best First Search. In: The 2014 International Planning Competition. pp. 39–42 (2014)

[66] Xing, E.P., Ng, A.Y., Jordan, M.I., Russell, S.: Distance Metric Learning with Application to Clustering with Side-Information. Advances in Neural Information Processing Systems 15, 505–512 (2003)

[67] Zickler, S., Veloso, M.: Efficient Physics-based planning: Sampling search via non-deterministic tactics and skills. Proceedings of the International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 1, 15–21 (2009)