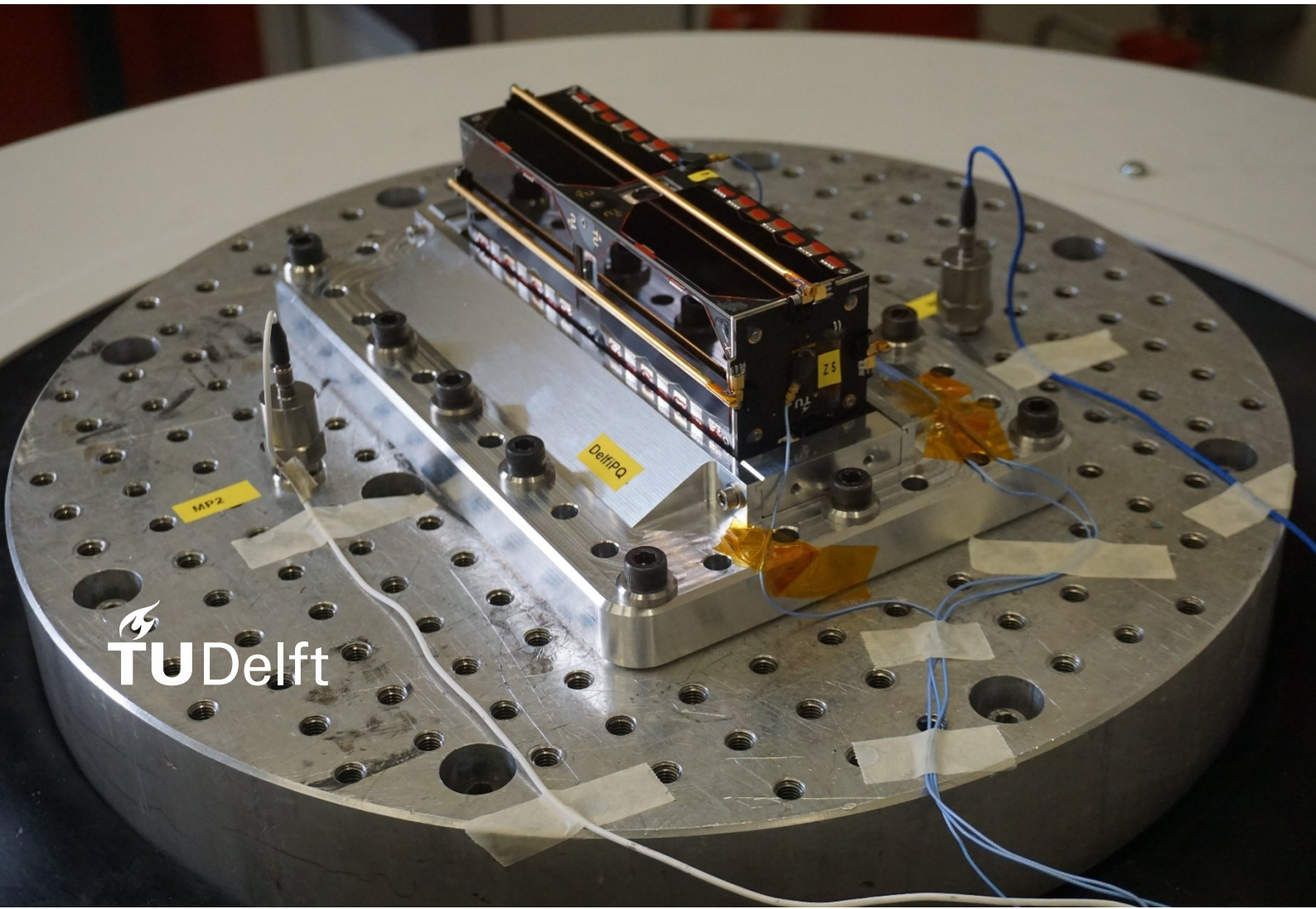
Planning and Reinforcement Learning for Delfi-PQ

A Literature Study

Zhuoheng Li



Planning and Reinforcement Learning for Delfi-PQ

A Literature Study

By

Zhuoheng Li

Student Number: 5003636
Supervisor: Dr. Alessandra Menicucci
Date: Dec 13, 2021
Version: 1.0

TU Delft

Contents

Abstract

This literature study investigates how to use the sequential decision-making techniques, i.e., planning or reinforcement learning, in the Delfi-PQ mission. Delfi-PQ is the first PocketQube satellite developed by TU Delft.

Given descriptions of the system, planning selects and organizes a sequence of actions that will eventually achieve goals set by human. Depending on the assumptions used, there are different types of planning problems: classical planning, temporal planning, probabilistic planning, non-deterministic planning, etc. Planning algorithms for these problems, history remarks and current trends in this field are reviewed in this document. To understand how planning techniques can be used for the Delfi-PQ satellite, the document also discusses previous applications of planning in the space industry, from the early 1980s to 2020s.

Reinforcement learning (RL) is another type of sequential decision-making technique which is getting popular in recent years. A reinforcement learning agent learns how to take the actions that maximize received rewards during interaction with the environment. Depending on how the agent represents the learned experience, reinforcement learning algorithms can be value-based, policy-based, model-based, or a mixture among them. The document reviews representative RL algorithms and briefly discusses current challenges in this field. It also briefly introduces neural networks, which are frequently used to construct components in RL agents.

Three possible applications of planning and reinforcement learning techniques are identified, including normal operation of the satellite, FDIR (Fault Detection, Isolation and Recovery), and automated test case generation for onboard software. Analysis shows that automated test case generation is the most useful and feasible application for Delfi-PQ.

After reviewing the existing test case generation techniques, a gap is identified. In current space industry, no system testing tool can efficiently generate causal-related command sequence with limited prior knowledge of the tested system. By contrast, such techniques based on the planning or reinforcement learning have become very mature in GUI testing and web service testing. Migration of such techniques from these fields to onboard software testing is still a challenge because of different environments.

Therefore, to reduce workloads of testers and improve the test coverage, we set the following research objective and related research questions as results of the literature study:

Suggesting an approach to automatically generate strong causal-related testing commands with limited prior knowledge for onboard software by designing and validation of a primitive planning/RL-based testing tool for the Delfi-PQ satellite.

1 Introduction

Sequential decision-making is a key challenge in artificial intelligence. Over the past decades, this problem has been studied by two different research communities: **planning** and **reinforcement learning** (Moerland, Broekens, & Jonker, 2020). Both communities developed their own conventions and successful applications.

There are also sequential decision-making problems for the Delfi-PQ, the first PocketQube satellite developed in TU Delft. This literature study focuses on how to apply planning and reinforcement learning techniques in the mission. This chapter first gives a brief introduction to planning, reinforcement learning, and the Delfi-PQ mission. At the end of the chapter, an overview of this document is given.

1.1. Planning

AI Planning is an important research field of Artificial Intelligence since the 1970s. It focuses on how to choose and organize future actions to achieve given goals (Ghallab, Nau, & Traverso, 2016). The following figure shows a simplified example of a planning problem.

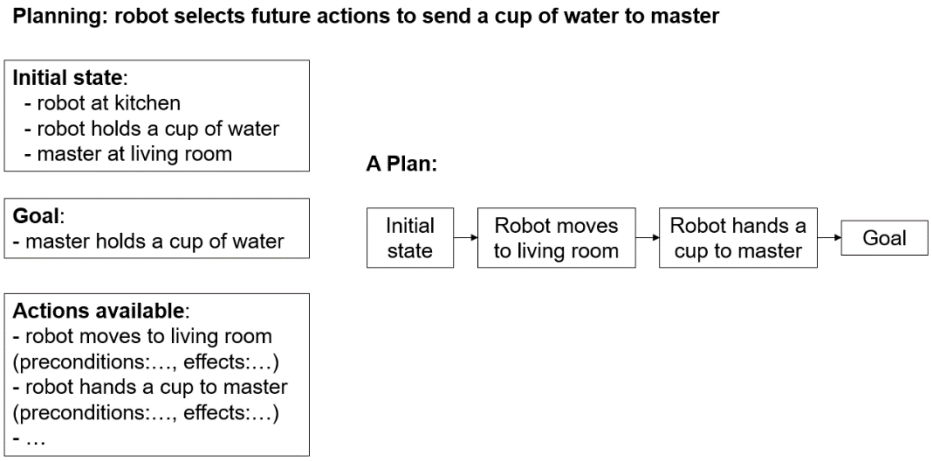


Figure 1: **Example of a Planning Problem**

Planning can be seen as a search process with prediction of future. Classical planning with several simplifying assumptions (see section 2.1) was the main topic in this area. Over the last 50 years, the search speed of classical planners has been improved by several orders of magnitude and can be used in some scenarios. Recent work in this area focuses on adapting planning to more complex scenarios (relax some assumptions), acquiring planning models automatically, and integration with other deliberation functions/agents (Ingrand & Ghallab, 2017).

1.2. Reinforcement Learning

Reinforcement learning (RL) is a computational approach to learn how to maximize the total amount of reward while interacting with a complex and uncertain environment (Sutton & Barto, 2018). Figure 2 shows an example of an RL problem.

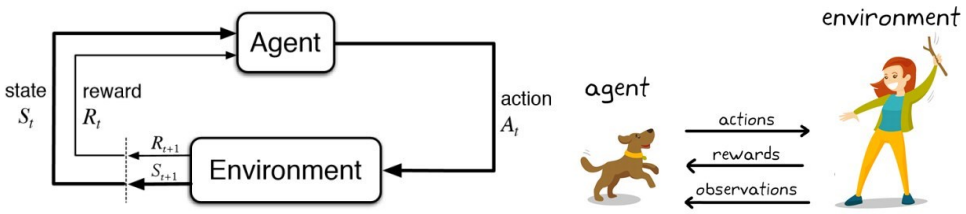


Figure 2: Example of a Reinforcement Learning Problem

From a view of search process, RL agents are learning heuristic functions. Model-based RL agents also learn searching models. A searching model can define a search space, such as “what nodes can be reach from the current node”. A heuristic function can guide the search process, like “which successor node is better”.

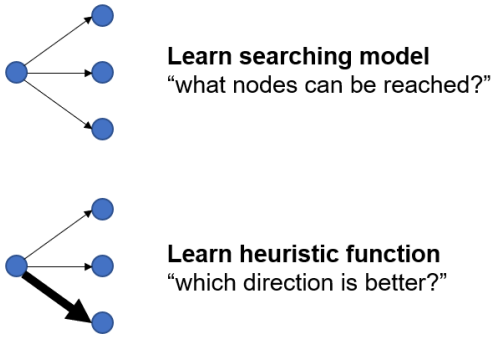


Figure 3: Learning of a Search Process

Knowledge of a search process can be learned from demonstration of expert, i.e., imitation learning (IL). RL and IL share many basic techniques. We will discuss both of them in chapter 3.

1.3. The Delfi-PQ Mission

After the Delfi-C3 launched in 2008 and the Delfi-n3Xt launched in 2013, the Delfi-PQ is the 3rd satellite build by Delft University of Technology (Radu et.al, 2018). It’s a 3P PocketQube (150x50x50mm). As a forerunner, Delfi-PQ helps to set a standard of this type of satellites (Bouwmeester, van der Linden, Povalac, & Gill, 2018). Most of subsystems, onboard software and electric ground support equipment are made in house.

Figure 4: **Delfi-PQ satellite**

This literature study looks for a way to apply planning and reinforcement learning in the Delfi-PQ mission. 3 possible applications are previously identified:

App1: Autonomous operation of the satellite.
App2: Autonomous recovery from faults.
App3: Automated testing of onboard software.

1.4. Overview of Rest of This Document

Chapter 2 reviews different types of planning problems and methods to solve these problems. It mostly focuses on classical planning and temporal planning since they are more often used in real-world applications. Some famous planners used in the space industry are briefly introduced. At the end of the chapter, it also gives a glimpse of the current trends in this field.

Chapter 3 reviews reinforcement learning. It mainly focuses on model-free RL algorithms because they are heavily researched. It also discusses topics like neural networks, Markov Decision Problems, model-based RL, hierarchical RL, imitation learning and meta learning. Some applications of reinforcement learning are briefly introduced.

Chapter 4 introduces objectives, hardware and software of the Delfi-PQ mission, and analyses where autonomy can be applied.

At the end of this document, chapter 5 proposes to apply planning and reinforcement learning techniques in the automated testing system of Delfi-PQ. It also suggests what can be improved compared with other methods and gives work packages of the research.

2 Planning

Figure 5 shows a view of planning in autonomous systems.

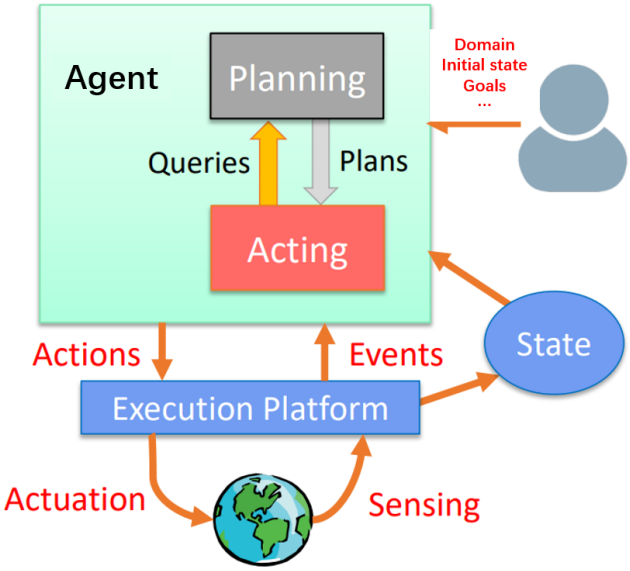


Figure 5: **Traditional View of Planning (Nau, 2021)**

In Figure 5, users input descriptions of the planning domain, and descriptions of the planning problem (the initial state s_0 and goals) into the planner. The planner makes plans. After that, the **acting module** refines actions in a plan into lower-level actions (commands), executes them and deals with uncertainty. The acting module also checks the execution status. If the status is unacceptable, it queries the planner to adapt the existing plan to new circumstances (**plan repair**) or generate a new plan from the current state (**re-planning**).

Planning can be (Nau, 2007)

- **Domain-specific**. These planners are specially designed and can only be used in a specific field, e.g., motion planning and path planning. The AI planning research field is less concerned with it because it highly depends on specific domains.
- **Domain-independent**. These planners can be applied in any field that satisfies some set of simplifying assumptions (see section 2.1). The sole input to the planner is a description of a planning problem to solve. Examples are the state space search, plan space search, GraphPlan, SAT Plan and CSP planners.
- **Domain-configurable**. The planning engine is domain-independent but the input to the planner includes domain-specific knowledge to constrain the planner's search. Examples are HTN planners and control-rule planners.

Planners with domain-specific knowledge are generally quicker than those without domain specific knowledge. However, it needs effort if users edit domain-specific knowledge by hand.

This chapter discusses domain-independent and domain-configurable planning. It first introduces common assumptions of planning. There are many types of planning problems based on different assumptions. Representations, languages, and algorithms for these problems are briefly reviewed, following by applications in the space industry and current trends.

2.1. Types of Planning Problems

Commonly used assumptions of planning problems are (Ghallab, Nau, & Traverso, 2004):

Assumption A0: the system has a finite set of states and actions.
Assumption A1: the system is fully observable.
Assumption A2: actions only have deterministic effects.
Assumption A3: only the actions of the agent can change the states, i.e., no external events.
Assumption A4: the goal is only a state or a set of states.
Assumption A5: a solution plan is a linearly ordered finite sequence of actions.
Assumption A6: actions have no durations.
Assumption A7: the planner is not concerned with any change that may occur during execution.

Classical planning problems satisfy all assumptions from A0~A7. It was most heavily researched and can be used in some simple scenarios.

Temporal planning relaxes A6 by including parallel actions and their durations. Its model is more realistic so many autonomous robot systems (e.g., spacecrafts) use it. Examples include the planning and scheduling module of the NASA Deep Space 1 spacecraft (Muscettola, Nayak, Pell, & Williams, 1998) and the APSI planning platform of ESA (Fratini, Cesta, De Benedictis, Orlandini, & Rasconi, 2011).

Probabilistic planning, where an action has multiple possible outcomes and the probability of each outcome is known, relaxes A2 and A5. Such problems can be modelled as Markov Decision Processes (MDP). A solution of probabilistic planning is a **policy** which maps states to actions, instead of a linear plan. The MDP approach has difficulties in integration with temporal planning.

In **nondeterministic planning**, outcome of actions is uncertain, and the probability is not fixed. One approach used to solve this problem is belief state search. A belief state is a set of world states, one of which is the actual state. This method relaxes A2. Another approach is conditional planning, whose solution plan is a tree structure with observation actions. This approach relaxes A2 and A5. Both approaches are not frequently used in practice.

A3 and A7 can be relaxed by **planning repair** or **replanning**. Replanning simply builds a new plan from scratch. In contrast, planning repair tries to reuse fragments of the original plan in the new plan.

There are many other types of planning problems which are not covered in this document. Note that:

- An algorithm may be able to solve different types of problems.
- In practical applications, there are not necessarily clear boundaries between the various types of planning problems. Different algorithms are also mixed.
- Several planning techniques can deal with uncertainty. However, in practice it is mainly the acting module that responds to uncertainty.

2.2. Representations

In the example of section 1.1, planning methods need to find a route from an initial state to a goal state in a state space. However, in most of planning problems, the state space is very large (NP-hard) and difficult to represent explicitly. Therefore, we need to find an implicit and compact way to

represent the state space. Such representation should enable us to generate the following things during planning:

- internal information about the current state
- actions that can be applied in the current state
- what state is reached after an applicable action is performed

There are 3 popular representations in classical planning (Ghallab, Nau, & Traverso, 2004):

- **Propositional representation**, which comes from the proposition logic. The world state is simply a set of propositions. Preconditions of an action are also propositions. Effects of an action are propositions that are added to, or removed from, the world state.
- **STRIPS representation**, which comes from the first-order logic. It was firstly introduced by the STRIPS planner. It has objects, variables (unknown objects), predicates (relations among objects), actions, and operators (actions with variables). A state consists of predicates with objects, also called ground atoms/literals. Preconditions of an action are literals. Effects of an action are literals that are added to, or removed from, the world state.
- **State-variable representation**. A world state consists of state variables. Actions are partial functions over states.

These 3 representations are equally expressive, which means a planning problem in one representation can also be represented by others. However, each representation may be more efficient than others in specific applications or algorithms. For example, the GraphPlan algorithm, which fully exploits the characteristics of the positional representation, was the most efficient planning algorithm in the mid-1990s.

Variants of these representations are used in other types of planning problems. In temporal planning, a variant of state variable representation called the **timeline representation** is commonly use (Ghallab, Nau, & Traverso, 2016). A timeline includes values of a state variable during different time intervals. For probabilistic planning and non-deterministic planning, all these 3 representations can be used with slight modification, i.e., there will be multiple possible outcomes of an action.

2.3. Languages

As shown by Figure 6, users need to input descriptions of the planning domain (the system $\sum$), and descriptions of the planning problem (the initial state s_0 and goals) into the planner. It is helpful to use standard languages to write definitions of planning domains and problems so different planners can have a standard interface.

Domain-independent planners for classical planning share a standard language now. It is called the Planning Domain Definition Language (PDDL), evolved from the STRIPS and ADL languages (McDermott, 2003). It was first developed for the international planning competition (IPC) in 1998 and becomes a complex language with many characteristics today. Most planers, however, do not support the full PDDL. The majority support only the STRIPS subset, or some small extensions of it.

Languages of domain-configurable planners for classical planning were also proposed. An example is the HDDL, an extension to PDDL for expressing HTN planning problems (Höller et al., 2020).

For probabilistic planning, there is an extension of PDDL 2.1 called the Probabilistic Planning Domain Definition Language (PPDDL), which supports probabilistic effects. It's now the standard planning language defined for the IPC probabilistic planning track (Younes & Littman, 2004). The NPDDL language for nondeterministic planning was defined in a similar way (Bertoli, Cimatti, Dal Lago, & Pistore, 2003).

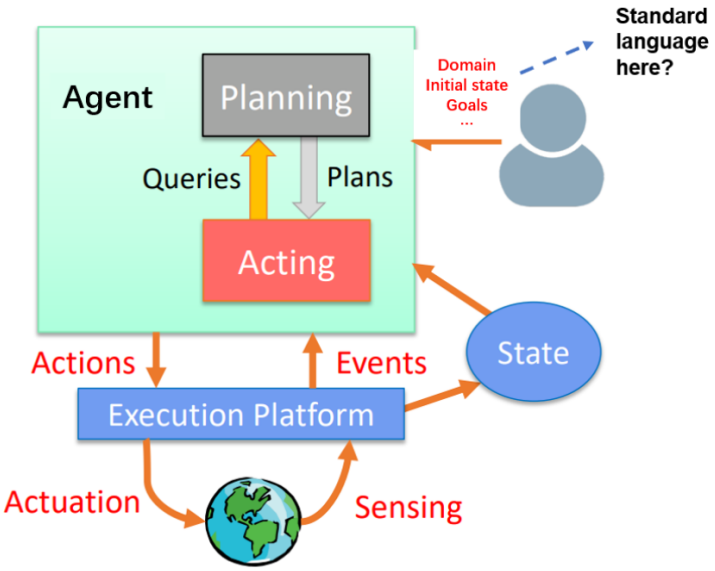


Figure 6: **Standard Languages Needed by Planning**

However, an extension of PDDL may not become the standard language in temporal planning. Although the current PDDL language provides some temporal characteristics, most of this language is based on the STRIPS representation, which is not flexible enough in temporal planning (we will discuss why in section 2.5). Languages of temporal planning need to support the popular timeline representation. Examples include the New Domain Definition Language (NDDL) of the Extensible Universal Remote Operations Planning Architectures (EUROPA) of NASA, and Domain Definition Language (DDL) used by the Advanced Planning and Scheduling Initiative (APSI) of ESA (Cividanes, Ferreira, & de Novaes Kucinskis, 2021).

2.4. Classical Planning

2.4.1 State Space Search

State space search is the most straightforward planning algorithm. It simply sees the planning process as finding a route from an initial state to a goal state on a state transition graph. As we mentioned in section 2.2, the state transition graph is implicit, and the planning algorithm constructs the nodes and edges of the graph on the fly.

State space search can be **informed** or **uninformed**. Uninformed search doesn't have any guidance. Examples of such methods include Depth-First search, Breadth-First search, and the famous Dijkstra's algorithm. Informed search uses **heuristic functions** to evaluate the distance between a state and the goal state. Examples of such methods include the Greedy Best-First Search (GBFS), the Depth-First Branch and Bound algorithm (DFBB) and A*-like algorithm. Practice shows that informed search can be more efficient than uninformed search.

State space search can also be forward search or backward search. **Forward search** starts from the initial state and tries to reach the goal states by repeating the following steps (backtracking and pruning are not shown here):

Step 1: Find applicable actions of the current state
Step 2: Choose an applicable action (informed/uninformed)
Step 3: Apply the action virtually and reach a new state
Step 4: Examine whether the new state is the goal state

By contrast, **backward search** starts from the goal state and try to reach the initial state by repeating the following steps (backtracking and pruning are not shown here):

Step 1: Find relevant actions of the current goal
Step 2: Choose a relevant action
Step 3: Find the state before applying the selected action. The new state is called a "sub-goal"
Step 4: Check whether the new sub-goal contains the initial state

Backward search generally has fewer branch number (branch factor) compared with forward search.

A technique called **lifting** can further reduce number of branches and therefore speed up state space search. It selects and applies operators (actions with variables), instead of fully ground actions that are applicable to the current state. In this way, it introduces variables into the state space and does not instantiate these variables if not necessary (least commitment principle). Therefore, this method reduces the branch number in the state space. However, as a price, it must manage the constraints among these variables. This idea is inherited by many other planning methods, especially the plan space search.

Some early planners used state space search. An example is the STRIPS planner (Fikes & Nilsson, 1971), which used a way like backward search. However, the state space is still too large even after backward search and lifting is used. Only with very good heuristic functions, state space search can reach a solution quickly. Over a long period of time, researchers didn't know how to come up with good heuristics.

This situation was changed by Bonet and Geffner (1999). They found that good heuristics can be easily computed by solving the relaxed problem, i.e., a simplified version of the original planning problem. It can be seen as generating domain-specific heuristics automatically.

This had led to fast planning algorithms like HSP (Bonet & Geffner, 1999) and FastForward (Hoffmann & Nebel, 2001). There are 4 types of advanced heuristics constructed in this way: **ignoring delete** effects on a planning graph (Keyder et al., 2012), **abstractions** (Helmert et al., 2007), **critical paths** (Haslum & Geffner, 2000), **landmarks** (Richter & Westphal, 2010). Some fastest planners today use forward state space search and these heuristics.

2.4.2 Plan Space Search

Before explaining the plan space search, we first introduce some basic concepts.

A **partial plan**, as shown by Figure 7, is an incomplete plan $\pi = (A, \prec, B, L)$, where:

- $A = \{a_1, \ldots, a_k\}$ is a set of partially instantiated operators, i.e., actions with variables.
- $\prec$ is a set of **ordering constraints** on A. $a_i \prec a_j$ means a_i is earlier than a_j.
- B is a set of **binding constraints** on the variables in A, such as $x = y$, $x \neq y$, and $x \in D_x$.
- L is a set of **causal links** $a_i - [p] \rightarrow a_j$ so that p is an effect of a_i and a precondition of a_j.

A partial plan becomes a **partial solution** if:

- no **open goal** (a type of **flaw**), i.e., every precondition in A is supported by a causal link.
- no **threat** (a type of **flaw**), i.e., no action that may interfere with a causal link $\prec$.
- the ordering constraints $\prec$ is not circular.

- the variable bindings B are consistent.

A partial solution can be linearized to total-order plans, i.e., sequences of actions that can convert the initial state to the goal state.

We can apply resolvers to a partial plan to make it closer to a partial solution. A **resolver** can be:

- adding an action
- adding a causal link
- add a variable binding
- add an ordering constraint

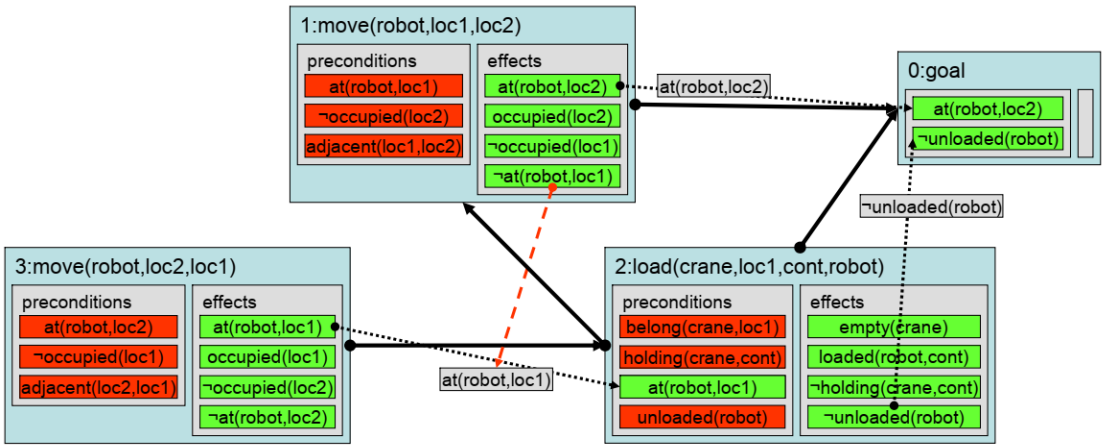


Figure 7: A Partial Plan (Wickler & Tate, 2012)
Solid lines are ordering constraints. Black dash lines are causal links. The red dash line is a threat. Variable bindings are not shown here.

In the plan space search, each node is a partial plan, and each edge is a resolver. We search in such plan space to find a way from a partial plan to a partial solution. The workflow of a partial space planner is shown in Figure 8.

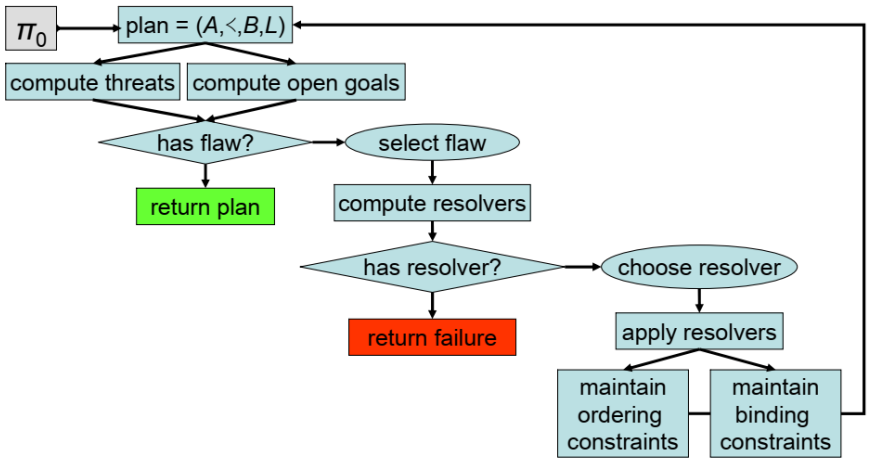


Figure 8: Workflow of Plan Space Search (Wickler & Tate, 2012)

In several planning domains, the plan space search is faster than the state space search. Another advantage is that rationale of the plan is explicitly represented by constraints and causal links. This makes plan repair easier. Therefore, when the plan space search was introduced by the NOAH (Sacerdoti, 1974) and NONLIN (Tate, 1977) planners in the late 1970s, it was a great breakthrough.

However, plan space search also has a significant problem: the current state of the system is always unknown. Without the current state, it's difficult to utilize domain-specific heuristics to guide the search. Therefore, the heuristic functions used by plan space search are less efficient than those used by state space search today. Examples of these heuristics are:

- Select the flaw with the smallest number of resolvers
- Choose the resolvers that rules out fewest resolvers of other flaws
- …

Computation in each node is also more expensive because such algorithms need to maintain ordering constraints and binding constraints. Sometimes it needs to solve a general Constraint Satisfaction Problem (CSP), which is NP-complete.

Therefore, the efficiency of plan space planners cannot be compared with advanced state space planners today, especially in classical planning. Some of its advantages are desired by planning with temporal and resources constraints because such planning problems must deal with constraints (Ghallab, Nau, & Traverso, 2004).

2.4.3 Hierarchical Task Network

In Hierarchical Task Network (HTN), there are tasks to decompose, instead of goals to achieve. We will first illustrate the basic idea of HTN using its simplest variants, total-order HTN planning.

Compared with the planning problem shown in Figure 2, a total-order HTN planning problem has some new concepts:

- **Tasks**. A primitive task can be performed by an action. A non-primitive task can be decomposed by one or several methods.
- **Methods**. A method is designed to decompose a non-primitive task. It has a sub task network and preconditions.
- **Task Networks**. In the total-order HTN planning, a task network is simply a sequence of task.

At the beginning of the planning process, an initial state, an initial task network, a set of actions, and a set of methods are given. The total-order HTN algorithm repeats the following steps until all non-primitive tasks are decomposed and all primitive tasks are replaced by actions (backtracking and pruning are not shown here):

Step 1: Select the first task in the current task network (a sequence).
Step 2: If the task is non-primitive, choose a method to decompose the task. The current state should satisfy the preconditions of the method. The non-primitive task is then replaced by the sub task network (a sequence) of the method.
Step 3: If the task is primitive, replace it with an action.

A plan may be found at the end. An example of a total-order HTN planning process is shown in Figure 9.

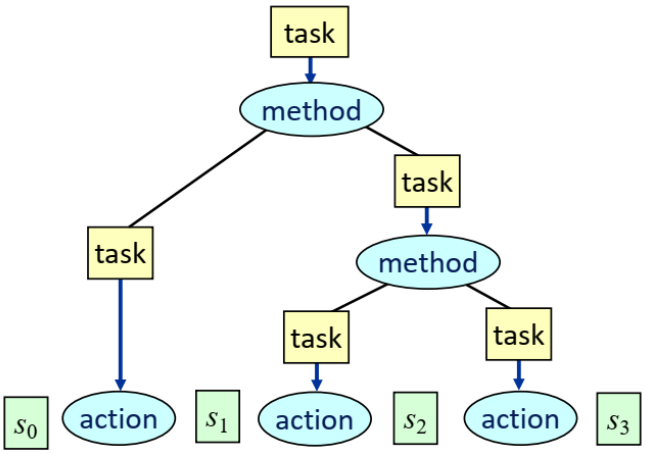


Figure 9: Illustration of Total-Order HTN Planning (Nau, 2021)

There are many variants of HTN Planning algorithms:

- **Total-order HTN planning** as shown here. The SHOP planner uses it (Nau, Cao, Lotem, & Munoz-Avila, 1999).
- **Partial-order HTN planning**, where a method only has a sub task network. A task network has tasks and constraints. A constraint looks like "precondition A is true, from the start of task B to the end of task C". The SHOP2 planner uses such algorithm (Nau et al., 2003).
- **Plan space HTN planning**, which combines HTN planning with plan space search. Examples are SIPE (Wilkins, 1990), O-Plan (Currie, & Tate, 1991), and UMCP (Erol, Hendler, & Nau, 1994).
- **Hierarchical Goal Network** (HGN). There is no task in HGN. HGN methods contain sub-goals, instead of sub task network. It can be seen as another type of hierarchical planning. An example is the GoDeL planner (Shivashankar, Alford, Kuter, & Nau, 2013).
- **Angelic Hierarchical A*** (AHA*). It does forward A* search at every level, from top down (Marthi, Russell, & Wolfe, 2008).

The advantage of HTN planning is thinking hierarchically. Different from state space search and plan space search, HTN doesn't look for a detailed plan at the beginning. It first finds a simple high-level plan and then define the details. This is exactly how human experts do their planning. At the same time, domain-specific search control knowledge is encoded in methods. If there are good methods, HTN planners can be several orders of magnitude faster than the classical planners (state/plan space search). Therefore, most of practical planners today use HTN planning.

The disadvantage is that HTN needs a set of good methods, which may take a lot of effort to develop. If at a certain state, no method can decompose a task, then the HTN algorithm will crash.

2.4.4 Planning with Control Rules

Another way to utilize domain-specific knowledge is **planning with control rules**. It uses linear temporal logic formulars to prune nodes during search. These formulars will be updated when the world state is changed. If there are good control rules, such planners can be several orders of magnitude faster than the classical planners. However, it needs good control rules written by human experts. An example of such planners is the TLPlan (Bacchus, & Kabanza, 2000).

It's interesting to compare control-rule planners with HTN planners. HTN tells the planner which parts to consider, while control rules tell the planner which parts not to consider. It's possible to integrate them together in some scenarios (Ghallab, Nau, & Traverso, 2004).

2.4.5 GraphPlan

A **planning graph** is a data structure such as the one shown in Figure 10. It uses the propositional representation in the section 2.2. Each level *n* contains propositions and actions that can be applied to propositions in level *n-1*. The graph also maintains **precondition/effect links** among layers, and **mutual-exclusive propositions/actions** in each layer. Thus, the planning graph represents a relaxed version of the planning problem in which several actions can appear simultaneously even if they conflict with each other (Nau, 2007).

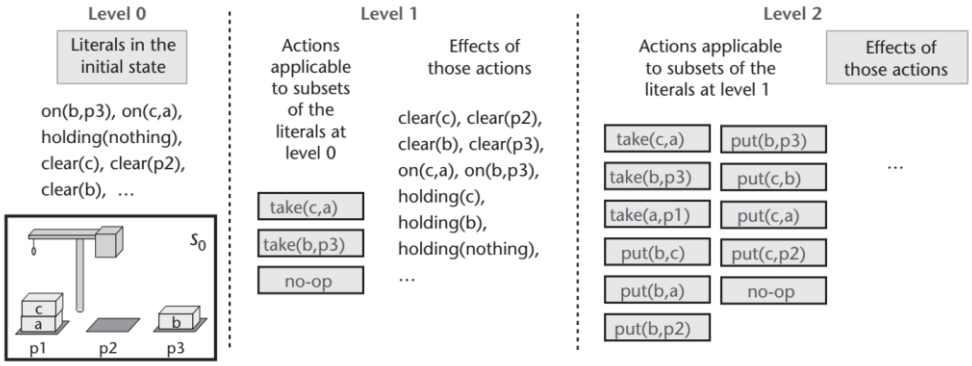


Figure 10: An Example of Planning Graph (Nau, 2007)

The basic GraphPlan algorithm repeats:

Step 1: Create a planning graph of *n* levels.

Step 2: Goals of the *n*-th level are the goals of the problem.

Step 3: If the *n*-th level contains all goals of this level and they are not mutually exclusive, look for actions in the *n*-th level that are not mutually exclusive and can satisfy all goals.

Step 4: Goals of the *n-1* level are preconditions of the action found in the previous step.

Step 5: Do similar things in step 3, but in level *n-1*.

Step 6: Repeat step 4 and 5 for level *n-2*, *n-3*, …, *1*. If it succeeds, it will find a plan.

Step 7: If step 6 fails, set *n* = *n+1* and go to step 1.

The planning graph can be expanded in polynomial time (step 1) and the mutually exclusive restrictions that the backward search must operate within the planning graph dramatically improve the efficiency of the backward search. As results, GraphPlan runs orders of magnitude faster than other domain independent planning algorithms. When it first appeared in the mid-1990s, it was a remarkable breakthrough in AI planning (Blum, & Furst, 1997).

It's popular to use the GraphPlan to generate heuristics instead of using it as a planner directly. The **relaxed planning graph** ignores "delete" effects of all actions so there will not be mutually exclusive propositions/actions. It simplifies the whole algorithm. By calculating the solution of such relaxed problem, a planner can better estimate the distance between a state and the goal, i.e., the heuristic function. An example is the FastForward planner (Hoffmann & Nebel, 2001).

2.4.6 Translation into Other Problems

A planning problem can be translated to combinatorial problems, such as satisfiability problems (SAT), constraint satisfaction problems (CSP), and integer programming. There are some efficient solvers that can deal with such problems. At the end, the solution found by the problem solver is translated to a plan. This approach leads to planners like SATPlan (Kautz, & Selman, 1999).

In fact, CSP techniques are often used to aid planning, rather than encoding planning problems directly into CSP problems. For example, such techniques can manage variable binding constraints in plan space search, or mutually exclusive propositions/actions in GraphPlan.

2.5. Temporal Planning

2.5.1 Planning and Scheduling

Traditional planning and scheduling are 2 different research fields. “Planning” decides which actions to take in an order to reach a desired goal, while “scheduling” allocates time and resources to some known actions (Hopgood, 1993).

In some simple scenarios, we can plan before we schedule. We first decide which actions to do and in what order, then allocate time and resources to each action. This approach simplifies both planning and scheduling, as shown by Figure 11.

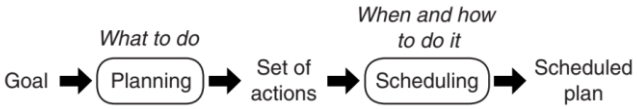


Figure 11: Simple Decomposition of Planning and Scheduling (Ghallab, Nau, & Traverso, 2004).

However, there may be inconsistency. For example, solutions from a planner may always violate temporal and resource constraints. To deal with such inconsistency, we need to integrate planning and scheduling together, i.e., temporal planning.

2.5.2 Temporal Planning Based on Durative Actions

Early temporal planners simply extend classical planning by giving each action a duration. According to PDDL2.1 (Fox, & Long, 2003), preconditions for a durative action can be true at the start of the action or be true all the way through the action. The effect of a durative action can occur at the start or end of the action. Figure 12 shows an example of such action.

In the durative action approach, the temporal model is not very complex. Therefore, classical planning techniques mentioned above can be reused here with modifications. Some efficient planners are designed in this way.

Many durative action planners use the state space search, and use heuristics improved from classical planning. An example is the SAPA planner (Do, & Kambhampati, 2003) which uses the relaxed time planning graph as its heuristic. Action compression (Eyerich, Mattmüller, & Röger, 2009), which abstracts the durative transition to an instantaneous one to calculate heuristic, is also popular.

A few planners with durative actions use the plan space search, like the ZENO planner (Penberthy, & Weld, 1994). Some planners also integrate HTN planning with durative action extension. Examples are the DEVISER (Vere, 1983), SIPE (Wilkins, 1990), O-Plan (Currie, & Tate, 1991), and SHOP2 (Nau et al., 2003).

Control-rule planners can also be extended with the durative action extension, such as the TALplanner (Kvarnström, & Doherty, 2000). It's also possible to encode such problems into SAT or CSP problems.

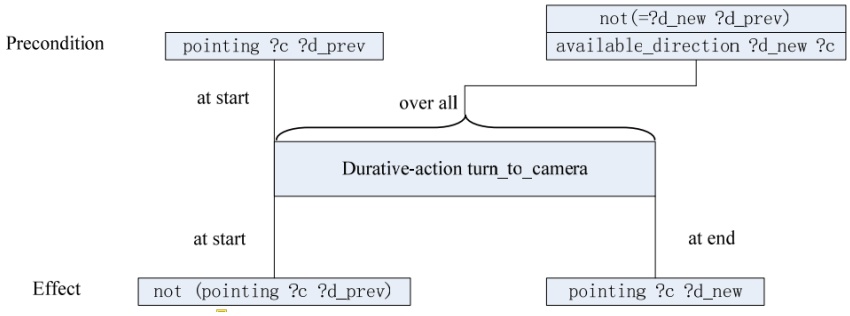


Figure 12: A Durative Action

Although the speed of the durative action approach is promising, the expressive of such model may be not enough in many applications. For example, such model cannot handle concurrent actions with interfere effects very well because these effects are instantaneous. Figure 13 shown an example of such case. In contrast, the timeline-based model in the next section can fix this problem.

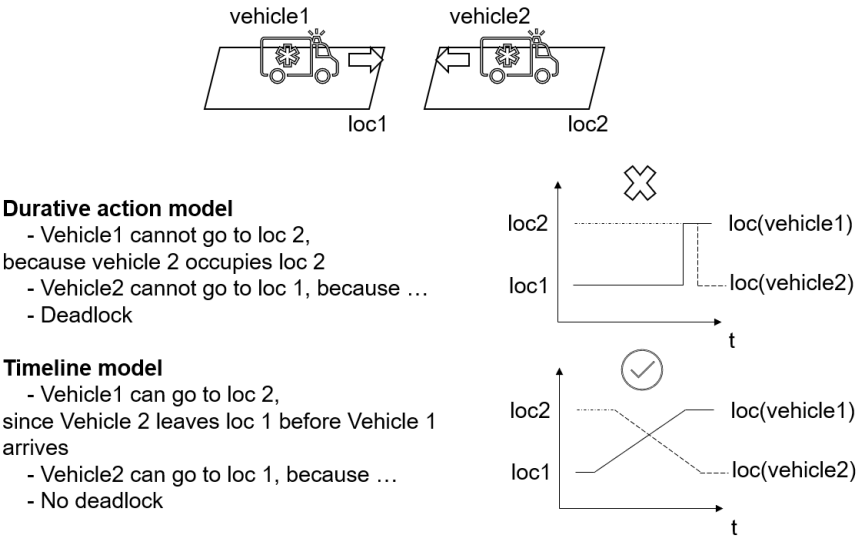


Figure 13: Concurrent Actions with Interfere Effects

2.5.3 Timeline-based Temporal Planning

Timeline-based planners use the basic ideas of the plan space search and HTN planning. Before explaining its ideas, we first introduce some basic concepts.

Each state variable has a **timeline**. We use the timeline in Figure 14 as an example, and it has:

- Timepoint variables (t_1, t_2, t_3, t_4), object variables (location l), real values (location loc1, loc2).

- Persistence temporal assertions, e.g., $[t_2, t_3]loc(r1) = l$. It shows that $loc(r1)$ remains to be l from t_2 to t_3.

- Change temporal assertions, e.g., $[t_3, t_4]loc(r1): (l, loc2)$. It shows that $loc(r1)$ changes from l to $loc2$ in the time interval $[t_3, t_4]$.

- A temporal assertion can be **causally supported** by another assertion. For example, $[t_3, t_4]loc(r1): (l, loc2)$ is supported by $[t_2, t_3]loc(r1) = l$. The concept of "causal support" is like the "causal link" in plan space search.

- Constraints on timepoint variables and object variables.

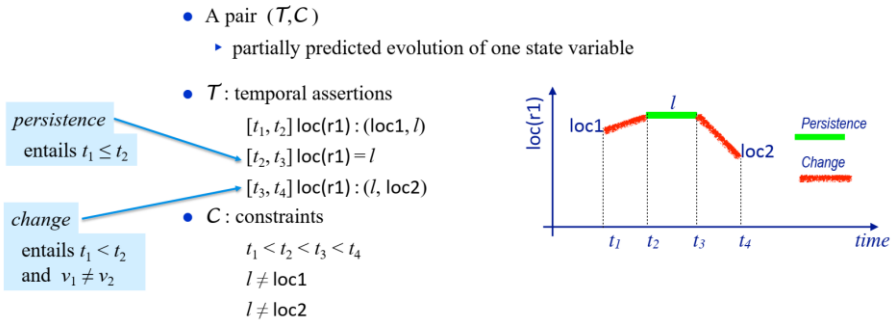
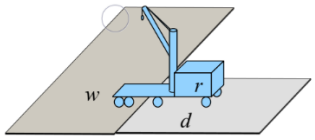


Figure 14: Timeline of State Variable loc(r1), Location of Robot r1 (Nau, 2021)

An **action** in the timeline representation doesn't have preconditions and effects anymore. Instead, it has temporal assertions and constraints, which will be inserted to timelines if the action is applied. Figure 15 shows an action in the timeline representation.



enter(r,d,w)

// robot r goes from waypoint w to loading dock d

assertions:

$[t_s,t_e]$ loc(r): (w,d)

$[t_s,t_e]$ occupant(d): (empty,r)

constraints:

$t_e \leq t_s + \delta_2$

adjacent(d,w)

Figure 15: An Action in Timeline Representation (Nau, 2021)

A **method** here also doesn't have preconditions. If a method is applied, new assertions and constraints will be added to timelines, and a composite task will also be decomposed to sub tasks.

m-bring(r,c,p,p',d,d')

task: bring(r,c,p)

refinement: [t_s,t_1] move(r,d') // task
[t_s,t_2] uncover(c,p') // task
[t_3,t_4] load(k',r,c,p')
[t_5,t_6] move(r,d) // task
[t_7,t_e] unload(k,r,c,p)

assertions: [t_s,t_3] pile(c) = p'
[t_s,t_3] freight(r) = empty

constraints: attached(p',d'), attached(p,d), $d \neq d'$
attached(k',d'), attached(k,d), $k \neq k'$
$t_1 \leq t_3$, $t_2 \leq t_3$, $t_4 \leq t_5$, $t_6 \leq t_7$

Figure 16: A Method in Timeline Representation (Nau, 2021)

A **chronicle**, as shown in Figure 17, is like a partial plan. It has:

- Current all primitive/composite tasks.
- Causally supported/unsupported temporal assertions of all timelines.
- Constraints from of timelines.

Like a partial plan, a chronicle will be a solution if:

- No unsupported temporal assertion (a type of **flaw**, like open goals in partial plans).
- No possibly conflicting temporal assertions (a type of **flaw**, like threats in partial plans).
- All composite tasks are decomposed. All primitive tasks are replaced by actions, and the assertions and constraints of these actions are added to the chronicle (a type of **flaw**).
- All constraints on timepoints are consistent.
- All constraints on object variables are consistent.

ϕ_2: tasks: [t_s,t_1] move(r;d1)
[t_s,t_2] uncover(c1,p′1)
[t_3,t_4] load(k1,r;c1,p′1)
[t_5,t_6] move(r; d3)
[t_7,t_e] unload(k3,r;c1,p3)
[t'_s,t'_1] move(r',d2)
[t'_s,t'_2] uncover(c2,p′2)
[t'_3,t'_4] load(k4,r',c2,p′2)
[t'_5,t'_6] move(r',d4)
[t'_7,t'_e] unload(k2,r',c2,p′2)

supported: [0] loc(r1)=d3
[0] loc(r2)=d4
[0] freight(r1)=empty
[0] freight(r2)=empty
[0] pile(c1)=p′1
[0] pile(c2)=p′2, . . .

assertions: [t_s,t_3] pile(c1) = p′1
[t_s,t_3] freight(r) = empty
[t'_s,t'_3] pile(c2) = p′2
[t'_s,t'_1] freight(r') = empty

constraints: $t_s<t_1\leq t_3$, $t_s<t_2\leq t_3$, $t_4\leq t_5$, $t_6\leq t_7$,
$t'_s<t'_1\leq t'_3$, $t'_s<t'_2\leq t'_3$, $t'_4\leq t'_5$, $t'_6\leq t'_7$
adjacent(d1,w12), . . .
connected(w12,w13) , . . .

Figure 17: A Chronicle in Timeline Representation (Nau, 2021)

Like in plan space search, we can define **resolvers** for a chronicle, which make it closer to a solution:

- Adding an action or a task
- Adding a persistence assertion
- Add a constraint on timepoint/general variables
- Decompose a composite task using a method
- Replace a primitive task using an action and add assertions & constraints of the action to the chronicle.

With these concepts, the workflow of timeline-based planners is like the one in Figure 8. It simply repeats the following steps until a solution is found (backtrack and pruning not shown here):

Step 1: Select a flaw.
Step 2: Select a resolver for the flaw.
Step 3: Apply the resolver and maintain constraints in the chronicle.

Like variable binding constraints of plan space search, constraints on objects are maintained as a general CSP. However, maintaining constraints on timepoint variables here are more complex than checking circular loops in plan space search. Such constraints are generally represented and managed by a Simple Temporal Network (STN) or the interval algebra.

The timeline approach was firstly introduced by Allen and Koomen (1983). The first planner that introduced chronicles was the IxTeT (Ghallab, & Laruelle, 1994). This approach is popular today, for example, most of planners for space industry in the section 2.7 are timeline-based. Compared with the durative action approach, it's less efficient but can deal with more complex situations. Because the timeline approach uses plan space search and keeps causal links and constraints, plan repair is also easier.

2.6. Planning with Uncertainty and Probability

In **probabilistic planning**, an action may have multiple outcomes, and the possibility of each outcome is known. It can be modelled as a Markov Decision Process (MDP) where state transition probabilities are known.

Early solvers of such problems used policy iteration or value iteration of dynamic programming. These solvers will calculate optimal policies of all states, so they take too much computational resources. Later algorithms avoid to evaluating all states. They focus on the states that can be reached from the initial state and use forward search based on AND/OR graphs.

This document doesn't dive into probabilistic planning. Such technique is difficult to use in the Delfi-PQ mission because we don't know the probabilities in advance. In the next chapter, we will see how probabilistic planning is used in model-based reinforcement learning.

Conditional planning doesn't need the probabilities. It produces plans with tree structures:

```
If <condition>
        plan_A
else
        plan_B
```

It also based on AND/OR graph search. The bad news is that conditional planning is harder than NP and use a lot of memory space O(2^n). In practice, we generally handle uncertainty with the acting module.

2.7. Planning in Space Industry

For some simple scenarios, where the spacecraft only has few actions to do, the approach in the section 2.5.1 is enough. For example, in normal operations of a remote sensing satellite, there will be only 2 actions: “turn to a target and take a photo”, and “downlink the data”. In this case, we can use a scheduler to make yearly/weekly schedules. Human operators will then translate the schedules to commands by modifying a set of command templates. The Hubble Space Telescope is operated in this way, with a CSP-based scheduler called SPIKE (Johnston, 1990). The author of this document also designed several scheduling algorithms for a China-ESA space telescope called the Einstein Probe (NAOC, 2018).

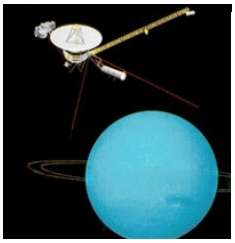
For more complex scenarios, where the spacecraft has many actions to select, organize and synchronize, temporal planning is needed. For example, a Mars rover needs to move, communicate, and perform multiple scientific operation. The Spirit and Opportunity rovers were supported by the MAPGEN ground planner (Ai-Chang et al., 2004), while an onboard planner was designed for the Perseverance rover (Rabideau et al., 2020).

There are many planners used in the space industry. The most representative examples among them are presented in this section. A conference in this field is the International Workshop on Planning and Scheduling for Space (IWPSS), organized by JPL. 2 surveys of planners in space industry are (Chien, 2012) and (Cividanes, Ferreira, & Kucinskis, 2019).

2.7.1 DEVISER (JPL, early 1980s)

DEVISER (Vere, 1983) mentioned in section 2.5.2 was one of the earliest planners in the space industry. In early 1980s, NASA JPL recognized that they needed temporal planning in deep space missions. As their first try, they developed DEVISER based on the NONLIN planner (Tate, 1977) mentioned in section 2.4.2.

The program is written in Lisp. It used plan space search and simple temporal constraints. The planner was tried when the Voyager spacecraft encountered Uranus but not actively used.



```
(ROLL ACTION
    (CONTEXT
        (NEQ refstar.1 refstar.2)
        (AZIMUTH refstar.1 star1.azimuth)
        (AZIMUTH refstar.2 star2.azimuth))
    ((SC.AZIMUTH star1.azimuth)
        (LOCKED.ONTO refstar.1 )) →
        ((SC.AZIMUTH star2.azimuth)
        (LOCKED.ONTO refstar.2))
     (DURATION (84 + 5 * (SCOPE.O.TO.180
        (star2.azimuth - star1.azimuth)))))
(GOALS ((A))
    ((WINDOW BETWEEN 15 300) (DURATION 1000) (B) (C))
    ((DURATION INFINITY) (D)))
```

Figure 20: Examples of Actions and Goals of the Voyager Mission (Clement, 2013)

2.7.2 Plan-IT (JPL, 1987-2001)

JPL’s Plan-IT ground planner (Shepperd et al., 1998) are improved versions of DEVISER. It used **local search**, which is also a type of plan space search. The plan space search mentioned before

starts from a blank initial plan and ends with a (not fully instantiated) partial plan without flaws. However, local search algorithms start with a ground plan with flaws, and end with a ground plan without flaws. Plan-IT also modelled depletable (like propellant) and non-depletable resources (like electrical power).

Plan-IT was used in many missions:

- It created activity plans for Galileo at Jupiter in 1994.
- From 1996 to 1997, it created science and operation plans for the Mars Sojourner rover.
- From 1998 to 2001. It generated all ground command sequences for the Deep Space 1 mission.
- This planner was also used to manage NASA's Deep Space Network.
- Creating plans for the Spitzer space telescope.
- Operate the DATA-CHASER payload on the space shuttle. In the DATA-CHASER mission (Shepperd et al., 1998), it was reported to reduce 80% of operation effort and increase 40% of scientific return.

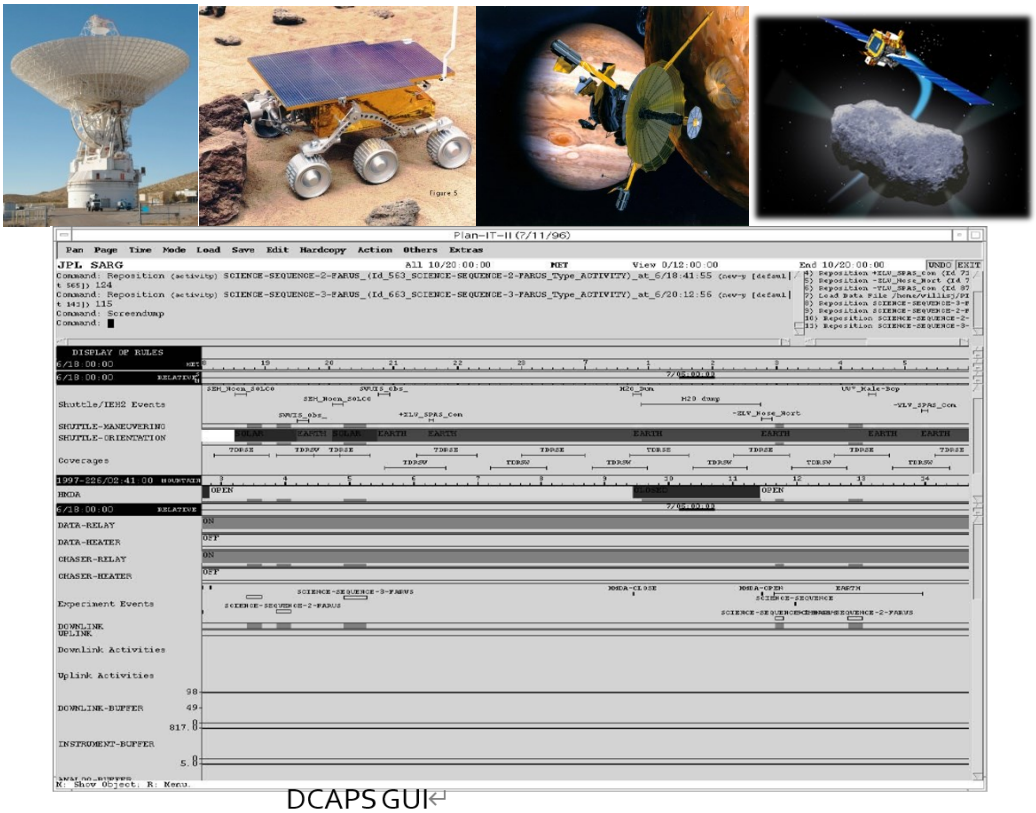


Figure 20: Plan-IT and Its Applications (Clement, 2013)

2.7.3 PS Module in DS-1 (NASA Ames, 1998)

The Planner/Scheduler module of Deep Space 1 (Muscettola et al., 1998) was the first onboard planner over the world. The architecture of the Deep Space 1 remote agent was exactly the architecture mentioned in chapter 1. Such architecture is also common in autonomous robots. The "Mission Manager" is a goal reasoning module. The "Planner/Scheduler" is a planning module. The "Smart Executive" is an acting module. The "livingstone MIR" is a FDIR module. The "monitors" are observing module. As an early agent, Deep Space 1 didn't have a learning module.

The Planner/Scheduler can also communicate with several external programs called “planning experts”. For example, an ADCS expert can calculate the time needed by attitude maneuver and send the result to the Planner/Scheduler. In this way, the complexity of planners is reduced.

The Planner/Scheduler of DS1 used the timeline approach in section 2.5.3. However, it didn’t integrate HTN methods inside. It also didn’t use local search but always started from a blank partial plan. As a result, it takes approximately 4 hours to produce a 3-day operations plan. Such long planning time is not desired in many missions.

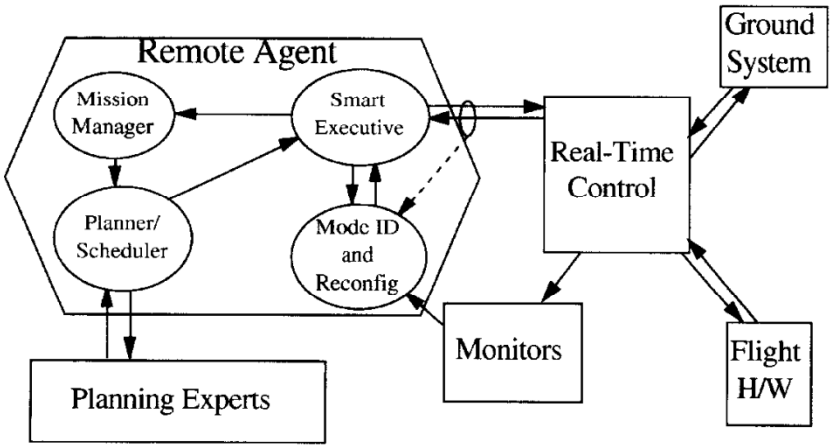


Figure 21: Architecture of Deep Space 1 Remote Agent (Muscettola et al., 1998)

The remote agent only controlled the Deep Space 1 spacecraft for 5 days (17th-21st May 1999). Some issues and alarms did arise during the experiment. These alarms mainly came from the complexity of the agent. They show that formal verification methods are needed for such mission-critical software.

2.7.4 ASPEN/CASPER (JPL, 2001-present)

The ASPEN system developed by JPL was another iterative repair local search planner like the Plan-IT. It was written in C++ and integrated HTN methods. Apart from the local search characteristic, it is very like the timeline approach in section 2.5.3.

An onboard version of ASPEN is called CASPER. CASPER doesn’t generate a plan from scratch but modify plan uploaded from the ground. Such plan repair only takes several seconds and is a remarkable improvement compared with the Planner/Scheduler of DS-1.

Its applications include:

- It was first used in the Modified Antarctic Mapping Mission and reported to reduce 80% planning effort and save more than $1M.
- ASPEN (ground) and CASPER (onboard) was used in the Earth Observation 1 satellite from 2004 to 2017, reducing 55% of operation costs.
- Since 2007, ASPEN is used to generate plans for the DARPA Orbital Express and save more than $10M. In this mission, ASPEN used nature language processing to generate planning model automatically.
- Since 2007, ASPEN is used to generate plans for the NASA Deep Space Network.
- In 2014, ASPEN and CASPER was also used in the IPEX CubeSat mission, where ASPEN generated a coarse plan on the ground and CASPER modified the plan onboard.

Figure 22: ASPEN and Its Applications (Clement, 2013)

2.7.5 EUROPA (NASA Ames, 2004-present)

The EUROPA planner is an improvement of the Planner/Scheduler module on DS-1. The program was rewritten in C++, instead of Lisp. It called timelines as “tokens”. It was used in

- Since 2004, as a part of MAPGEN, it generated plans for the Spirit and Opportunity mars rovers.
- Since 2007, as a part of SACE, it planned the orientation and movements of the International Space Station’s eight large solar arrays.

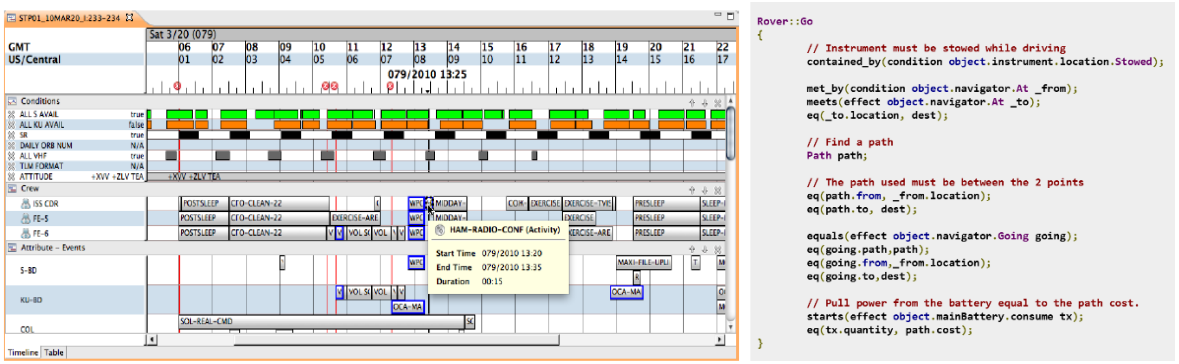


Figure 23: EUROPA Planner (Clement, 2013)

2.7.6 MEXEC (JPL, 2019-present)

MEXEC is developed by JPL (Troesch et al., 2020). It was first tried in the ASTERIA CubeSat and run within 2MB of memory (the number of actions is limited). MEXEC includes a HTN planner, an executive (acting module), a state database and a timeline library. The timeline library can manage temporal and object constraints and calculate valid intervals for tasks. It leaves an easy-to-use interface to the planner. This feature is popular on the recently developed temporal planners.

The onboard planner of the Perseverance Mars rover (Rabideau et al., 2020) was specially designed but shared the same timeline library of MEXEC. MEXEC is multi-mission, while the onboard planner only has a specialized planning algorithm with limited choice points.

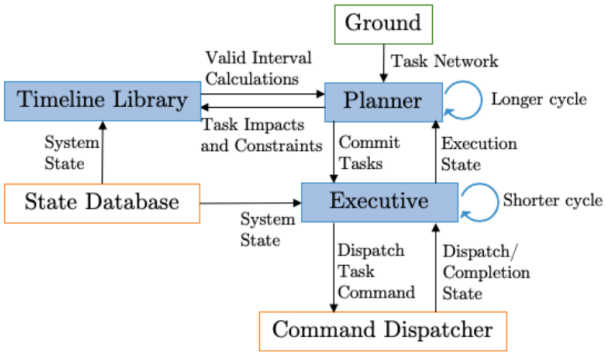


Figure 24: Architecture of MEXEC (Troesch et al., 2020)

2.7.6 APSI (ESA, 2006) and European Planners

APSI (Fratini et al., 2012) is a platform to support planners and schedulers. It manages timelines and constraints for planners / schedulers, which is like the timeline library of MEXEC. Several schedulers, like the downlink scheduler MAXER of Mars Express, were developed on APSI.

ESA is also working on integrating the OPTIC timeline-based planner (Benton & Coles, 2012) into their Mars rover. ESA also developed planners including:

- PlanERS, which was the earlies ESA planner (Fuchs, Gasquet, Olalainty, & Currie, 1990).
- Optimum-AIV, which was developed from the Edinburgh O-Plan mentioned in section 2.4.3 (Arentoft, Parrod, Stader, Stokes, & Vadon, 1991). It was used in project management of the assembly, integration, and verification (AIV) of the vehicle equipment bays of the Ariane 4 rockets.

There are many other European planners, for example, the Flexplan of GMV.

2.8. Current Trends in Planning

We will discuss some important directions in the following subsections.

2.8.1 Dynamic and Open World

Most of classical planners assume the environment is fully observable (A1 in section 2.1), static (A3), and planners are not concerned with any change that may occur during execution (A7). However, these assumptions are not true in many applications.

In most of applications, the acting module will handle dynamic and partially observable environment as much as possible. However, to reduce the complexity of the acting module, the planning module needs some capability to handle dynamic and open environment.

Plan repair and replanning.

It has been widely accepted. For example, the CASPER system in section 2.7.4 takes this approach. (Rui, Chen, Cui, Zhu, & Xu, 2019) surveys current plan-repair techniques:

- Use rules to identify why the current plan fails and fix the problem. It's simple but repair rules may not cover all situations. Examples are the CHEF (Hammond, 1990) and O-Plan system (Drabble, Dalton, & Tate, 1997).

- Use historical plans to modify current plans. It's quick but takes a lot of memory space to store historical plans and cannot guarantee a solution. Researchers like Fukushima and Mita (2011) try to improve its efficiency.

- Delete the failure action and try to find action(s) which can replace the failure action. If not possible, delete more actions from the plan and try again. Examples are (Gerevini, & Serina, 2000), (Van Der Krogt, & De Weerdt, 2005) and (Mohalik, Jayaraman, Badrinath, & Feljan, 2018).

- Repair the failure action. An action fails because some of its preconditions are not true. Such algorithms try to find actions that can make these preconditions true. An example is (Guzman, Castejon, Onaindia, & Frank, 2015)

- Retrieve information from the original plan and use such information as "soft goals" of the new planning problem. An example is (Talamadupula, Smith, Cushing, & Kambhampati, 2013).

Generally, if the original plan contains more information about "why this action is placed here", plan repair will be easier. If plan repair is not possible, we must do replan which starts from scratch. Replan takes more time and may lead to inconsistency.

There are many research questions to answer in this area:

- How to evaluate feasibility of plan repair before we start to do it?
- How to minimize the time and resources needed by plan repair? There has been a lot of work done in this question.
- Determine which part of the plan are affected by the failure action, so we can only fix that part.
- Evaluate stability of a plan.
- Plan repair on concurrent actions and their conflicts on time and resources. This is important for operation of spacecraft.

Generate a flexible plan.

We have seen this approach in temporal planning, probabilistic planning (where a plan is a policy), and conditional planning (where a plan is a tree structure). The Planner/Scheduler of DS-1 takes this approach and always generates a partial plan with some variables. For example, the timepoint variables in a DS-1 plan are not substituted by real values. The "Smart Executive" will set these timepoints according to constraints in the plan during acting.

Recent developments of flexible temporal plans include research on the Simple Temporal Network with Uncertainty (STNU). In temporal planning, we generally assume an action has certain known duration. However, duration of an action can be longer or shorter in practice. A STNU is **dynamically controllable** if we can find a policy that guarantees success regardless of actual duration of an action (Hunsberger, Posenato, & Combi, 2012). A STNU with such policy can be seen as a conditional temporal plan. Some techniques like the graph neural network can be used to analyse the dynamically controllability of STNU (Osanlou, 2021).

Probabilistic planning is not frequently used in practice because we don't know the probability in most of time. In some cases, where most of actions are deterministic and only few of them are probabilistic (Likhachev, Thrun, & Gordon, 2004), probabilistic planning is applicable.

Better prediction

If the environment is uncertain, why not just improve the prediction capability of the planning module? AI planning generally uses "precondition-effect" representation of actions, which is too shallow but has high searching speed (Ingrand and Ghallab, 2017). By contrast, **sampling search** determines outcomes of an action by simulation, which can provide good prediction but take a lot of time.

A compromise can be a reinforcement learning approach. The agent is first trained under sampling search (the simulation may be accelerated by another neural network). After training, the agent can select actions without explicit sampling because such experience is stored in its neural network (Patra, Mason, Kumar, Ghallab, Traverso, & Nau, 2020). However, this approach is currently limited by bad generalization of reinforcement learning, i.e., if the goals or domain definitions are changed, the agent needs to be trained again.

2.8.2 Integration with Other Deliberation Functions

Apart from planning, an autonomous agent also has other deliberation functions (Ingrand & Ghallab, 2017), including acting, learning, observing, monitoring and goal reasoning:

Acting: Agent acts during execution of a plan. It focuses on how to perform an action now, instead of choosing what action occurring in future. It decomposes the current action in the plan into low-level commands, executes them and deals with environments with uncertainty.

Observing: The agent needs to observe signals and extract current state of the system from signals. However, it is not simply a data processing process in complex systems. For example, a satellite may have thousands of parameters in its telemetry. What parameters need to be paid attention to when certain problems occur? How can we deal with relations among these parameters, e.g., data fusion? To solve these problems, some reasoning capability is needed.

Monitoring: It's also called Fault Detection, Isolation and Recovery (FDIR). It detects the discrepancies between the prediction and current observations, diagnoses the causes of such discrepancies, surveys whether the current goals are still feasible and if not, find a way to solve it (e.g., set new goals for the planner).

Goal Reasoning: It sets goals for planning. Sometimes human give many goals over a long period (e.g., 1 week) to an autonomous agent. Therefore, the agent needs to decide which goal is needed in the next planning horizon (e.g., 1 hour).

These deliberation functions also use different knowledge models so there may be inconsistency. There are researchers who want to integrate several deliberation functions in a single model, i.e., using similar or the same knowledge model and representation in different deliberation functions.

The integration of planning and monitoring is quite successful. By monitoring some state variables called **planning invariants**, it's possible to monitor whether the current plan can reach the goal or not. We can also use model-based diagnosis approach, or the temporal action logic mentioned in section 2.4.4 to monitor whether the current plan is feasible or not (Ingrand and Ghallab, 2017).

Integration of planning and acting is possible by using some HTN-like algorithms (Ghallab, Nau, & Traverso, 2014). The work of Patra et al. (2020) also integrates planning, acting, and learning.

The integration of planning and observation remains to be a challenge. An autonomous robot needs to plan sensing actions and focus on which part of telemetry. It also needs some types of reasoning capabilities to identify relations among telemetry parameters. Such integration can be a promising direction.

2.8.3 Learning

There are two approaches to achieve learning in planning. One approach is learning a neural network, which can represent heuristic function (policy/value network in RL) and domain definition

(model-based RL). Recent research also considers causal-representation learning. We will discuss this approach in the next chapter.

Another approach is **symbolic learning**, which learns symbolic heuristics and domain definitions from demonstration or trials. Symbolic learning relies on techniques like inductive logic programming (ILP). They are not very popular because of errors and limited complexity in learned models. 2 surveys of this approach are (Zimmerman & Kambhampati, 2003) and (Jiménez, De La Rosa, Fernández, Fernández, & Borrajo, 2012).

We can also transfer other symbolic models to planning models or require a planning model from a semantic web (Ingrand & Ghallab, 2017). For example, the RoboEarth (Waibel et al., 2011) and KnowRob (Tenorth & Beetz, 2013) projects build platforms to share and reuse planning knowledge over the internet. They share models of objects (e.g., images, CAD models), environments (e.g., maps and object locations), and actions together with their relations and properties in a general ontology.

Current planning research is mainly based on hand-written symbolic knowledge. Future planning may be based on some automated learned knowledge representations that have enough complexity and good generalization. It may reform the planning area.

3 Reinforcement Learning

This chapter mainly focuses on reinforcement learning (RL):

- Section 3.1 will introduce basics of artificial neural networks because they can be used as components of an RL agent.
- Section 3.2 will briefly introduce the Markov Decision Process.
- Section 3.3 will introduce value-based RL algorithms. It focuses on the Deep Q Network (DQN). This section will also briefly review variants of DQN.
- Section 3.4 will introduce policy-based RL algorithms. It introduces the policy gradient algorithms. This section will also briefly review its variants, including TRPO, PPO, ACER, AKTER, A3C/A2C, DDPG, TD3 and SAC.
- Section 3.5 briefly reviews current challenges of reinforcement learning, including large amount of sampling, sparse or hard to define rewards, and migrations to different tasks. Several approaches to solve these challenges are introduced, including off-policy, model-based RL, imitation learning (IL), hierarchical RL, curiosity mechanism, multi-task RL and meta-learning.
- Section 3.6 introduces 3 famous RL agents of DeepMind, including the AlphaGo Zero, MuZero and AlphaStar.

3.1. Basics of Artificial Neural Networks

3.1.1 Structure

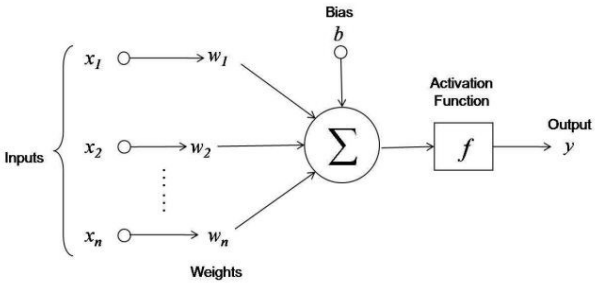


Figure 25: An Artificial Neuron

An artificial neuron has multiple inputs x_i $(i = 1, \dots, n)$ from other neurons or external world. The sum of the inputs is

$$s = \sum_i w_i x_i + b \tag{3-1}$$

Here w_i are **weights of connections** and b is the **bias**. The output of the neuron is

$$y = f(s) \tag{3-2}$$

And f is the **activation function**. There are many types of activation functions, like sigmoid and ReLU.

An artificial neural network consists of neurons and their connections with weights. The simplest neuron network is the Back-Propagation Neural Network (Rumelhart, Hinton, & Williams, 1986), as shown by Figure 26.

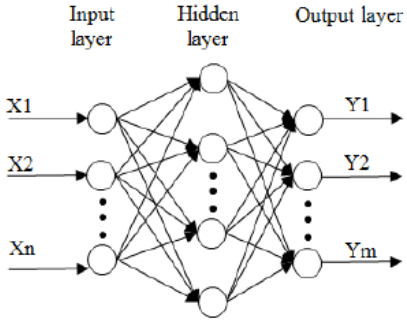


Figure 26: Structure of BP Neural Network

In the BP network, output of each layer is forwarded to the nodes in the next layer. The input vector $[x_1, \dots, x_N]^T$ is sent to the input layer, and we get an output vector $[y_1, \dots, y_M]^T$ from the output layer. Given an input vector, a neural network should be able to minimize a **loss function**.

For a **classification** task, each sample s has a feature vector $[x_{1s}, \dots, x_{Ns}]^T$, which is input of the neural network, and a result vector $[p_{1s}, \dots, p_{Ms}]^T$, where p_{ms} is a binary indicator (0 or 1) indicating the correct classification for sample s for class m. The loss function can be a cross entropy

$$Loss = -\frac{1}{S}\sum_{s=1}^{S}\sum_{m=1}^{M} p_{ms} log(y_{ms}) \tag{3-3}$$

where M is the number of classes, S is the number of samples, and y_{ms} is the predicted probability in sample s for class m. By minimizing the loss function, a neural network fit the relation between feature vectors and result vectors.

For a **regression** task, the loss function can be a mean squared error:

$$Loss = -\frac{1}{S}\sum_{s=1}^{S} \|\boldsymbol{p_s} - \boldsymbol{y_s}\|^2 \tag{3-4}$$

Where $\boldsymbol{p_s}$ is the result vector of a sample, and $\boldsymbol{y_s}$ is the predicted vector from the neural network. Here again, by minimizing the loss function, a neural network fit the relation between feature vectors and result vectors.

Apart from the BP network, there are many network structures designed for different tasks, like the Convolution Neural Network (CNNs), Recurrent Neural Network (RNNs), Transformers, etc. New network structures are proposed every day.

3.1.2 Training

Training of a neural network is an optimization process. We change weights (and sometimes structures) of the network to minimize the loss function. This can be achieved by the gradient descent method.

The first step is to find the partial derivatives of the loss function with respect to each of the weights $\frac{\partial Loss}{\partial w_i}$. This can be done by a process called **back propagation**. For example, in Figure 27 there are 3 neurons whose outputs are y_1, y_2, y_3. There are also 3 connections whose weights are w_1, w_2, w_3. If we know $\frac{\partial Loss}{\partial y_3}$, it's easy to calculate:

$$\begin{aligned}\frac{\partial Loss}{\partial y_2} &= \frac{\partial Loss}{\partial y_3}\cdot\frac{\partial y_3}{\partial y_2} = \frac{\partial Loss}{\partial y_3}\cdot\frac{\partial f(w_2y_2)}{\partial(w_2y_2)}\cdot w_2 \\ \frac{\partial Loss}{\partial y_1} &= \frac{\partial Loss}{\partial y_2}\cdot\frac{\partial y_2}{\partial y_1} = \frac{\partial Loss}{\partial y_2}\cdot\frac{\partial f(w_1y_1)}{\partial(w_1y_1)}\cdot w_1\end{aligned} \tag{3-5}$$

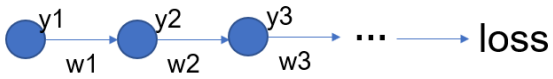


Figure 27: Demonstration of Back Propagation

So, the back propagation (Rumelhart, Hinton, & Williams, 1986) simply means that we can calculate the partial derivatives of the current layer using the partial derivatives of the previous layer. With all $\frac{\partial Loss}{\partial y_i}$, we can compute $\frac{\partial Loss}{\partial w_i}$:

$$\begin{aligned}\frac{\partial Loss}{\partial w_2} &= \frac{\partial Loss}{\partial y_3}\cdot\frac{\partial y_3}{\partial w_2} = \frac{\partial Loss}{\partial y_3}\cdot\frac{\partial f(w_2y_2)}{\partial(w_2y_2)}\cdot y_2 \\ \frac{\partial Loss}{\partial w_1} &= \frac{\partial Loss}{\partial y_2}\cdot\frac{\partial y_2}{\partial w_1} = \frac{\partial Loss}{\partial y_2}\cdot\frac{\partial f(w_1y_1)}{\partial(w_1y_1)}\cdot y_1\end{aligned} \tag{3-6}$$

The second step is to use the partial derivatives to update the weights, for example:

$$w_i = w_i - \alpha\frac{\partial Loss}{\partial w_i} \tag{3-7}$$

where α is the learning rate of gradient descent. There are many variants of the gradient descent methods. For example, the mini-batch gradient descent only uses part of samples in the loss function. Some new optimization algorithms like Adam and RMSdrop also rely on gradient information from back propagation.

3.1.3 Limitation

A deeper neural network with more layers can fit more complex relation. However, as a neural network gets deeper, researchers found the **vanishing gradient problem**. If the partial derivatives are less than 1, from eqn (3-5, 3-6) we can image the $\frac{\partial Loss}{\partial w_i}$ of deeper layers will become smaller. Eventually, $\frac{\partial Loss}{\partial w_i}$ in some layers will be so close to 0 that eqn (3-7) won't update weights w_i in these layers anymore. The vanishing gradient problem limits the depth (number of hidden layers) of neural networks for a long time. Nowadays, this problem can be mitigated by simpler activation functions like ReLU ($max(0, x)$), carefully designed network structures (ResNet, batch normalization, etc), and smarter training methods like dropout.

Samples should be **independent and identically distributed** (i.i.d). If samples are related or following different distributions, it will be difficult for a neural network to "grasp" the relation between result vectors and feature vectors of the samples.

An inherent problem of current neural network is that it can only fit a relation. As the relation becomes more complex, the neural network needs to have more parameters and layers. OpenAI's GPT-3 model (Brown, 2020) already has 175 billion parameters. Shall we keep increasing the size of the model to fit almost all relations in this world? A possible solution may be adding some kinds of reasoning capability (like symbolic techniques in chapter 2) in neural networks. For example, causal representation learning can not only find relations, but also causal links among entities (Schölkopf et al., 2021).

A brief survey and future directions of neural network research can be found in (Bengio, Lecun, & Hinton, 2021).

3.2. Markov Decision Process

3.2.1 Concepts in Markov Decision Process

A Markov decision process is a 4-tuple (S, A, P, R), where (Wikipedia, 2021)

- S is a set of states called the state space.
- A is a set of actions called the action space. A_s is the set of actions available from state S.
- $P(s'|s,a)$ is the probability that action a in state s will lead to state s'.
- $R(s'|s,a)$ is the **immediate reward** received after transitioning from state s to state s', due to action a.

In a Markov Decision process, the state transition probability is only related to the current state and action, instead of the entire history of the agent's interaction with the environment. This assumption simplifies the problem a lot and makes search in a MDP model like search on a graph.

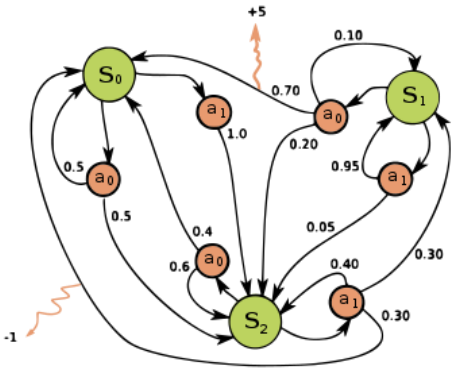


Figure 28: Example of a MDP Model

In **episodic problems**, there are special states called **terminate states**. If an agent enters such state, an episode will finish, and the length of the episode is called the **horizon** T. There aren't terminate states in **continuous problems**, where the length of an episode $T = \infty$ (Sutton, & Barto, 2018).

A **return function** G_t is the cumulated reward from time step t to horizon T:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \tag{3-8}$$

Where γ is the **discount factor**. If $\gamma = 1$, the return function will be very high since horizon T can be very large. In this case, the return is almost the same regardless of the agent's choice of action. To avoid this, we generally set $\gamma \in (0,1)$, which shows long-term rewards are less important than short-term rewards.

A **policy function** $\pi(a|s) = P(a|s)$ determines the probability that the agent will select action a in state s. Under a policy π, an **action-value function** $Q^{\pi}(s,a)$ shows the expected return of state s if action a is selected (it's time-independent):

$$Q^{\pi}(s,a) = \mathbb{E}[G_t|s_t = s, a_t = a] \tag{3-9}$$

Under a policy π, a **state-value function** $V^{\pi}(s)$ shows the expected return of state s (time-independent):

$$V^{\pi}(s) = \mathbb{E}[G_t|s_t = s] = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a) \tag{3-10}$$

3.2.2 Estimate Value Functions by Sampling

In reinforcement learning, researchers estimate value functions (3-9) and (3-10) by sampling. There are 2 types of sampling methods: Monte Carlo Sampling and Temporal Difference Sampling.

The basic idea of the **Monte Carlo sampling** (MC) is to acquire several complete trajectories $\tau = (s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, \dots, r_T, s_T)$ from state s_t and action a_t under policy π. Each trajectory has a return function, and we use the average of these return functions as an estimation of $Q^{\pi}(s,a)$. $V^{\pi}(s)$ can be also computed using eqn (3-10).

The tabular version of Monte Carlo sampling maintains a Q table in which estimations of all $Q^{\pi}(s,a)$ are stored. The initial state of the sampling is randomly selected. After obtaining a trajectory under policy π, from estimation $\hat{Q}^{\pi}(s_T, a_T)$ to $\hat{Q}^{\pi}(s_1, a_1)$ in the Q table are updated by repeating:

$$\begin{gathered} \hat{Q}^{\pi}(s_t,a_t) \leftarrow \hat{Q}^{\pi}(s_t,a_t) + \alpha\left[G_t - \hat{Q}^{\pi}(s_t,a_t)\right]; \\ t \leftarrow t-1; \\ G_t = r_{t+1} + \gamma G_{t+1}; \end{gathered} \tag{3-11}$$

Figure 29: A Q Table to Store $\hat{Q}^{\pi}(s,a)$

α in (3-11) is the learning rate of Monte Carlo sampling. If $\alpha = \frac{1}{N(s_t)}$ where $N(s_t)$ is the total number that state s_t is visited, it means all trajectories affect the Q table equally. If α is a constant, new experience will have more influence on the Q table.

Under the Monte Carlo sampling, the estimation of $Q^{\pi}(s,a)$ has no bias but high variance. Variance is introduced because trajectories from the same initial state can be very different. Another disadvantage is that the Monte Carlo sampling can only be used in episodic MDP problems. Otherwise collecting a complete trajectory will be impossible because a trajectory is infinite long.

Different from the Monte Carlo sampling, the **temporal difference sampling** (TD) can use incomplete trajectories. For example, the single-step TD(0) only uses trajectories like $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1})$ and update the Q table with

$$\hat{Q}^{\pi}(s_t, a_t) \leftarrow \hat{Q}^{\pi}(s_t, a_t) + \alpha\left[r_{t+1} + \gamma \hat{Q}^{\pi}(s_{t+1}, a_{t+1}) - \hat{Q}^{\pi}(s_t, a_t)\right]; \tag{3-12}$$

There are also multi-step TD, as shown in Figure 30.

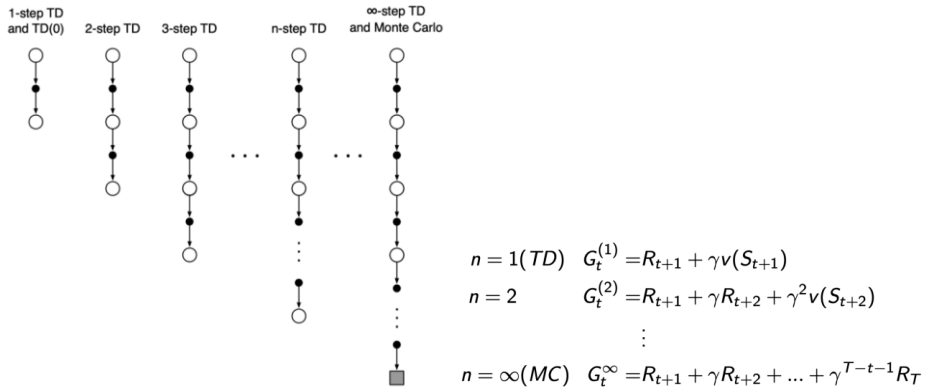


Figure 30: Multi-Step Temporal Difference Sampling

Under the Temporal Difference sampling, the estimation of $Q^{\pi}(s, a)$ has bias and low variance. Bias is introduces because the estimation $\hat{Q}^{\pi}(s_{t+1}, a_{t+1})$ in (3-12) is usually not accurate. Compared with the Monte Carlo sampling, it has lower variance because it only uses shorter, incomplete trajectories.

3.3. Value-Based Reinforcement Learning

Value-based RL algorithms have explicit action-value functions, but they don't have explicit policy functions. Their policies can be calculated using the action-value functions in some ways.

In this section, we will use simple tabular methods like SARSA and Q-learning to illustrate the basic ideas behind value-based RL. Then we will explain how neural networks can do the similar things better.

3.3.1 SARSA and Q-Learning

We start from the simplest MDP problem called the multi-Arm bandit problem, as shown in Figure 31. There are only 3 actions a_1, a_2, a_3 and one state s in this example. Each a_i provides a reward (get money or not) from an unknown probability distribution p_i. We assume p_i and $Q(s, a_i)$ remain the same regardless of how an agent interact with the bandit.

A (tabular) value based RL agent maintains a Q table. In this case, the Q table has only one row and three columns. Since the agent's objective is to maximize the expected return, its policy should be choosing the action with the highest $\hat{Q}(s, a_i)$. However, as we see from Figure 31, the initial values of $\hat{Q}(s, a_i)$ are incorrect. If a policy randomly chooses actions and updates the Q table with $\hat{Q}(s, a_i) \leftarrow \hat{Q}(s, a_i) + \alpha\left[G_i - \hat{Q}(s, a_i)\right]$ for many times, the agent will get good estimation for $Q(s, a_i)$.

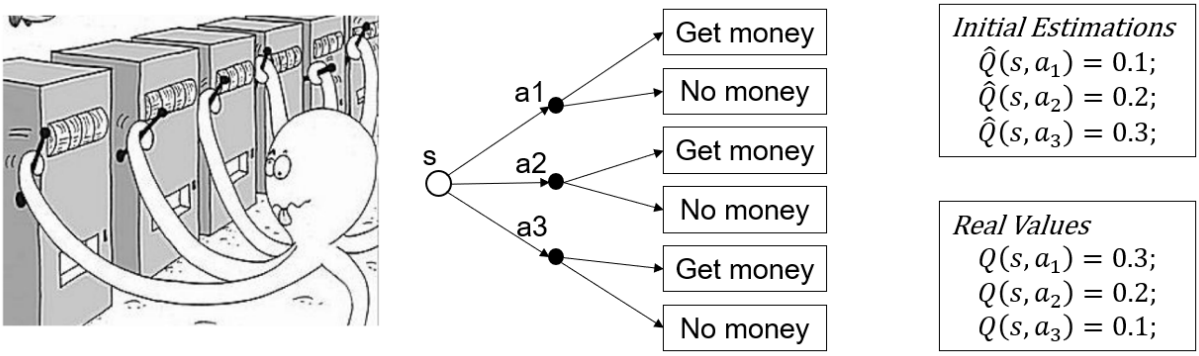


Figure 31: Value-Based Reinforcement Learning on Multi-Arm Bandit

It's called the **exploitation vs exploration dilemma**. When the Q values are inaccurate, we hope the policy chooses action that's less familiar. When the Q values are accurate enough, we hope the policy chooses the action that has given highest rewards in the past. In practical problems, we don't know whether these Q values are accurate or not.

One solution is to use a slightly changing policy. An example is the **ε-greedy policy**. In such policy, the agent selects the action a_i with the highest $\hat{Q}(s, a_i)$ with probability 1 – ε, or it selects an action randomly with probability ε. ε is a small value (e.g., 0.1) and usually decreases over time. In this way, the RL agent explores more space at the beginning and believes more in the results of learning. If an RL agent always uses a slightly changing policy, it's called **on-policy** approach.

By contrast, the **off-policy** approach uses 2 policies, the **behavior policy** for training and the **target policy** for real application. The behavior policy usually allows more effective exploration of the state space, while the target policy greedily chooses the best action. The key is how to use samples from the behavior policy to evaluate the target policy. If such evaluation is not done properly, the training process will be unstable (Sutton, & Barto, 2018).

Fortunately, in the multi-Arm bandit example, $Q(s, a_i)$ remains the same regardless of what policy the agent takes. It's possible to use a behavior policy to estimate the $Q(s, a_i)$, and use the estimation $\hat{Q}(s, a_i)$ in the target policy directly. The 2 approaches will be:

- In the on-policy approach, we simply use an ε-greedy policy with a slightly decreasing ε. Values in the Q table will converge to real values.
- In the off-policy approach, we can use a random behavior policy and a greedy target policy. Values in the Q table will converge more quickly because the behavior policy does more exploration than the ε-greedy policy.

From this simple example, we know the offline approach can reach optimal performance in a shorter time, at the cost of a potentially unstable training process.

Now we move forwards to more realistic MDP problems, as shown by Figure 32. The main difference compared with the multi-Arm bandit case is that $Q^{\pi}(s, a)$ will change if the policy π changes. For the ε-greedy policy, $Q^{\pi}(s, a)$ will change with the parameter ε.

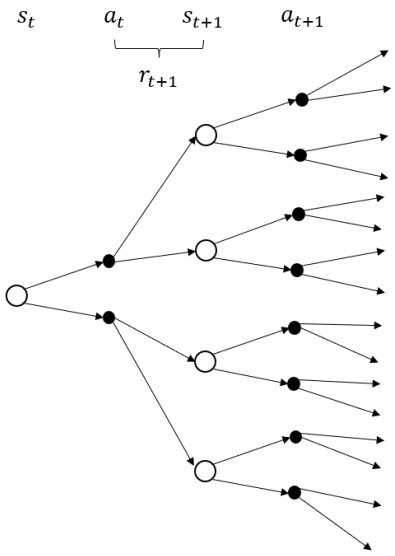


Figure 32: A more realistic MDP

An on-policy algorithm called **SARSA** (Rummery, & Niranjan, 1994), will keep sampling the state space using the ε-greedy policy with a slightly decreasing ε. During sampling, it updates its Q table using eqn(3-12). We can image that:

- If ε is fixed and $Q^{\pi}(s,a)$ remains unchanged, the Q table will reach good estimations of $Q^{\pi}(s,a)$, as it does in the multi-Arm bandit case.
- After that, ε will slightly decrease and the Q table will reach good estimations of $Q^{\pi}(s,a)$ under a new policy.
- (Repeating…)
- Finally, the Q table will reach good estimations of $Q^{\pi}(s,a)$ under a policy with $\varepsilon = 0$.

Obviously, such a greedy policy will be optimal if the estimation $\hat{Q}^{\pi}(s,a)$ is accurate.

An off-policy algorithm called **Q-Learning** (Watkins, & Dayan, 1992) have an ε-greedy behavior policy and a greedy target policy. During training, it uses the samples of the behavior policy to estimate $Q^{\pi}(s,a)$ of the target policy:

$$\hat{Q}^{\pi}(s_t,a_t) \leftarrow \hat{Q}^{\pi}(s_t,a_t) + \alpha\left[r_{t+1} + \gamma \cdot max\hat{Q}^{\pi}(s_{t+1},a) - \hat{Q}^{\pi}(s_t,a_t)\right] \quad (3\text{-}13)$$

Note that in (3-13) π means the fully greedy target policy, and $max\hat{Q}^{\pi}(s_{t+1},a)$ is the maximal $\hat{Q}^{\pi}(s_{t+1},a)$ under the state s_{t+1}. Q-Learning is proved to converge to the optimal if:

- ε approaches 0 not too quickly.
- Each state-action pair is visited infinitely often.
- It uses a Q table to store all estimations $\hat{Q}^{\pi}(s,a)$, instead of using a function or a neural network to fit these estimations.

Compared with the fully random behavior policy in the multi-Arm bandit example, the ε-greedy policy is more suitable like MDPs in Figure 32. A fully random policy will not pay more attention to a high-reward region in a large state space, but an ε-greedy policy will do so.

The ε-greedy policy in Q-Learning is also “bolder” in exploration than that one in SARSA. Assume the agent samples a very low $\hat{Q}^{\pi}(s_{t+1},a_{t+1})$, it will affect $\hat{Q}^{\pi}(s_t,a_t)$ in SARSA but not affect $\hat{Q}^{\pi}(s_t,a_t)$ in Q-Learning. After that, the SARSA agent doesn’t “dare” to take the action a_t in state s_t,

while the Q-Learning agent would still like to do so. Such property of Q-Learning may make it more efficient in exploration.

The SARSA and Q-Learning algorithm shown here use the single-step TD sampling. In fact, some variants also use multi-step TD sampling or even Monte Carlo sampling.

3.3.2 Deep Q-Learning (DQN)

If there are many states and actions, the size of a Q table will be so large to fit in any memory. To solve this problem, DeepMind (Mnih et al., 2015) proposed to use a neural network called **Deep Q Network** (DQN) to fit all estimations $\hat{Q}^{\pi}(s,a)$, as shown in Figure 33.

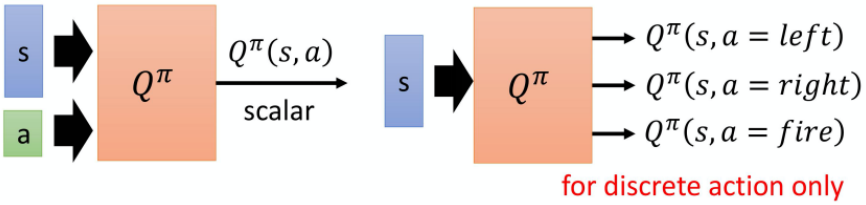


Figure 33: Two Types of Deep Q Networks

However, life is not so easy. They found many problems with this approach. A fatal problem is that the training process is very unstable.

The first reason of such instability is that the output from the neural network cannot converge to $\hat{Q}^{\pi}(s,a)$. According to (3-4) and (3-13), the loss function of such neural network should be

$$Loss = -\frac{1}{N_{\tau}}\sum_{\tau}\left[r_{t+1} + \gamma \cdot max\hat{Q}^{\pi}(s_{t+1},a) - \hat{Q}^{\pi}(s_t,a_t)\right]^2 \tag{3-14}$$

Here, N_{τ} is the number of incomplete trajectories $\tau = (s_t, a_t, r_{t+1}, s_{t+1})$ collected by the agent. From the view of a regression task, $r_{t+1} + \gamma \cdot max\hat{Q}^{\pi}(s_{t+1},a)$ is the result value of a sample, (s_t, a_t) is the feature of a sample (input of the neural network), and $\hat{Q}^{\pi}(s_t,a_t)$ is the predicted value (output of the neural network).

Minimization of (3-14) is difficult because the distribution of $\hat{Q}^{\pi}(s_{t+1},a)$ is changing all the time. We calculate $\hat{Q}^{\pi}(s_{t+1},a)$ using the neural network, but weights in the network is changed by the back propagation. Therefore, distribution of these samples is changing, which violates the i.i.d assumption of neural networks (see section 3.1.3).

Mnih et al. (2015) alleviated this problem by using a fixed but not accurate $\hat{Q}^{\pi}(s_{t+1},a)$. More specifically, they use two neural networks in their design: one is the current network to calculate $\hat{Q}^{\pi}(s_t,a_t)$, the other is the **target network** to calculate $\hat{Q}^{\pi}(s_{t+1},a)$. In N time steps, the target network is fixed and only the current network is updated. After that, the target network will be set equal to the current network. The whole process is shown in Figure 34.

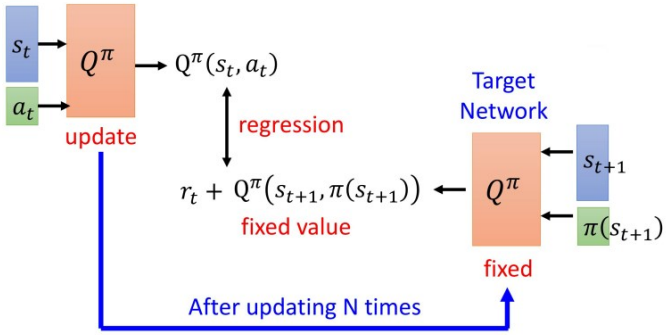


Figure 34: Current and Target Networks of DQN

The second reason of instability is that samples from recent N_τ trajectories $(s_t, a_t, r_{t+1}, s_{t+1})$ are highly related. DeepMind plays video games in their research. In such games, sample used to train the neural network comes from related frames close in time. Such relation violates the i.i.d assumption (see section 3.1.3) again and destabilises the training process.

Mnih et al. (2015) alleviated this problem by a large buffer to store trajectories, which is called **experience replay**. The agent randomly takes some trajectories $(s_t, a_t, r_{t+1}, s_{t+1})$ from the replay buffer. These trajectories are far apart in time and not related. They may come from different behavior policies and are used to evaluate different target policies. If the replay buffer is not too large and ε decreases slowly, such difference is acceptable.

Now the loss function becomes:

$$Loss = -\frac{1}{N_\tau}\sum_{\tau}\left[r_{t+1} + \gamma \cdot max\hat{Q}^{\pi}_{w^-}(s_{t+1}, a) - \hat{Q}^{\pi}_{w}(s_t, a_t)\right]^2 \quad (3\text{-}15)$$

N_τ trajectories are randomly collected from the replay buffer. $\hat{Q}^{\pi}_{w^-}(s_{t+1}, a)$ is calculated by the target network with weights w^-, which has a lag in weight updates. $\hat{Q}^{\pi}_{w}(s_t, a_t)$ is calculated by the current network with weights w. The DQN algorithm with these tricks is shown in Figure 35.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Figure 35 DQN Algorithm in (Mnih et al., 2015)

Given all these tricks, DQN is still not very stable during training. It's some kinds of inherent property of value-based RL algorithms because small updates to $\hat{Q}^{\pi}(s, a)$ may significantly change the policy

and therefore change the data distribution. Error of neural networks makes the situation even worse. Assuming $Q^{\pi}(s_1, a_1) = 0.5$ and $Q^{\pi}(s_1, a_2) = 0.51$, the best action under state s_1 should be a_2. However, because of error, the neural network may give $\hat{Q}^{\pi}(s_1, a_1) = 0.52$ and $\hat{Q}^{\pi}(s_1, a_2) = 0.51$. Such minor error makes training more instable and the target policy sub-optimal (**policy degradation**).

Although DQN is not perfect, it is able to solve many problems. DeepMind's DQN agent achieved better scores than human beings in the Atari 2600 video games. It was also a good beginning of deep reinforcement learning. Some other problems of DQN will be addressed in the next section.

3.3.3 Variants of DQN

DQN and Q-Learning tend to **overestimate** $Q^{\pi}(s, a)$. Since $\hat{Q}^{\pi}(s, a)$ from both the neural networks are very noisy, when we take max over all $\hat{Q}^{\pi}(s, a)$, we are probably getting an overestimated value (Lgvaz, 2017). For example, the expected value of a dice roll is 3.5, but if we throw the dice 100 times and take the max over all throws, we're very likely taking a value that is greater than 3.5.

A solution is the **Double DQN** (Van Hasselt, Guez, & Silver, 2015), whose loss function is

$$Loss = -\frac{1}{N_\tau}\sum_{\tau}\left[r_{t+1} + \gamma \cdot \hat{Q}^{\pi}_{w^-}\left(s_{t+1}, argmax\hat{Q}^{\pi}_{w}(s_{t+1}, a)\right) - \hat{Q}^{\pi}_{w}(s_t, a_t)\right]^2 \quad (3\text{-}16)$$

The trick is how to $max\hat{Q}^{\pi}(s_{t+1}, a)$ in (3-14). In (3-15), it's simply computed by the target network, i.e., $max\hat{Q}^{\pi}_{w^-}(s_{t+1}, a)$. In the Double DQN, we select the action with the maximal $\hat{Q}^{\pi}(s_{t+1}, a)$ using the current network and recalculate its $\hat{Q}^{\pi}(s_{t+1}, a)$ using the target network. Since 2 network has different weights, it is unlikely that they overestimate the same action. However, double DQN is more vulnerable to noise.

The $\hat{Q}^{\pi}(s_t, a_t)$ from DQN is only an estimation of the expected return G_t under state-action pair (s_t, a_t). The G_t is a random variable. **Distributional DQN algorithms** like C51 (Bellemare, Dabney ,& Munos, 2017), QR-DQN (Dabney, Rowland, Bellemare, & Munos, 2018) and IQN (Dabney, Ostrovski, Silver, & Munos, 2018) will estimate the distribution of G_t, instead of an expected value. Distributional DQN records more information of G_t to make "smarter" decisions. However, such algorithms are complex.

DQN is good at large state space, but **not good at large/continuous action space**. In eqn(3-15), how can we know which action makes $\hat{Q}^{\pi}_{w^-}(s_{t+1}, a)$ maximal, if there are too many actions? There are some tricks of DQN to fix such problem. However, policy based RL usually performs better at large/continuous action space, for example, the DDPG algorithm in section 3.4.4.

Prioritized Experience Replay (Schaul, Quan, Antonoglou, & Silver, 2015) doesn't randomly take trajectories from the buffer to train the network. Instead, it chooses the trajectories that lead to higher loss functions to update the network. It significantly increases the efficiency of DQN.

Dueling DQN (Wang et al., 2016) estimates $Q^{\pi}(s, a)$ by 2 networks: one estimates the state-value function $V^{\pi}(s)$, and the other estimate the **advantage function** $A^{\pi}(s, a)$. Estimations from these networks can be combined:

$$\hat{Q}^{\pi}(s_t, a_t) = \hat{V}^{\pi}(s_t) + \hat{A}^{\pi}(s_t, a_t) \quad (3\text{-}17)$$

In this case, the loss function (3-15) is calculated by 4 networks: $\hat{V}_{w^-}^{\pi}$, $\hat{V}_{w}^{\pi}$, $\hat{A}_{w^-}^{\pi}$, and $\hat{A}_{w}^{\pi}$. Such decomposition makes the estimation more accurate.

Noisy Net (Fortunato et al., 2017) gives up the ε-greedy behavior policy. Instead, it explores the state space by adding Gaussian noise to weights of its neural networks at the beginning of each episode. After that, new noise will not be added, and the agent will explore greedily until reaching a terminate state. Noisy net has better performance because for a DQN, it's quicker to estimate $Q^{\pi}(s,a)$ if the policy π doesn't include random actions.

DQN from demonstration (Hester et al., 2018) puts some pre-prepared good trajectories in the replay buffer and never deletes them. Obviously, the agent will learn in a shorter time. However, if the pre-prepared trajectories are too few or contain noise, the algorithm may overfit or be instable.

The **rainbow** algorithm (Hessel et al., 2018) simply combines all improvements mentioned above.

3.4. Policy-Based Reinforcement Learning

A policy based RL algorithm has an explicit policy function π, whose input is current state. For discrete action space, a policy function directly estimates the probability to take each action. We will discuss continuous action space in section 3.4.4.

The performance of a policy function is commonly evaluated by the objective function (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017):

$$J = \mathbb{E}_{(s,a)\sim w}[A^{\pi}(s,a) log \pi_w(a|s)] \tag{3-18}$$

$A^{\pi}(s,a)$ is the advantage function and shows how good action a is, comparing with other actions under state s. $\pi_w(a|s)$ is the probability to take action a under state s, with current policy parameters w. Generally, a policy π_w is a neural network with weights w. The state action pairs (s,a) can be seen as random variables, whose distribution is determined by the policy π_w.

A policy π_w with higher J is better. Therefore, the key idea of policy based RL algorithms is to estimate the policy gradient $\nabla_w J$,

$$\nabla_w J = \mathbb{E}_{(s,a)\sim w}[A^{\pi}(s,a) \nabla_w log \pi_w(a|s)] \tag{3-19}$$

and use gradient ascent method to maximize J. This section will introduce some classical policy-based algorithms.

3.4.1 REINFORCE

REINFORCE (Williams, 1992) uses the Monte Carlo sampling. After getting N complete trajectories $(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, \dots, r_{T_n}, s_{T_n})$, it estimates the gradient without bias:

$$\nabla_w \hat{J} = \frac{1}{N}\frac{1}{T_n}\sum_{n}^{N}\sum_{t=1}^{T_n} \hat{A}^{\pi}(s_t,a_t) \nabla_w log \pi_w(a_t^n|s_t^n) \tag{3-20}$$

$$\hat{A}^{\pi}(s_t,a_t) = G_t^n - b(s_t^n) \tag{3-21}$$

where $b(s_t^n)$ is the average G_t under the state s_t^n. After that, REINFORCE updates the weights w by

$$w \leftarrow w + \beta \nabla_w \hat{J}_w \tag{3-22}$$

Where β here is the update rate.

The main problem of REINFORCE is the low sampling efficiency, i.e., it needs too many samples to reach the optimum. After it updates the weights w (only one backpropagation), it must sample again to estimate the new gradient under new weights. Sampling takes most of training time.

Different from the Q-Learning, REINFORCE doesn't guarantee to reach optimum. It usually finds a local maximum. The estimation of the advantage function (3-21) also has high variance because of the Monte Carlo sampling.

3.4.2 Trust Region Methods (TRPO/PPO/ACER/ACKTR)

Trust region methods use samples of an old policy $\pi_{w'}$ to estimate policy gradient of the current policy π_w. According to **importance sampling** in statics, the gradient is:

$$\nabla_w J = \mathbb{E}_{(s,a)\sim w'}\left[\frac{\pi_w(s,a)}{\pi_{w'}(s,a)} A^{\pi}(s,a) \nabla_w log \pi_w(a|s)\right] \tag{3-23}$$

In (3-19), the probability distribution of the random variable (s,a) is determined by the current policy π_w. In (3-22), the probability distribution is determined by the old policy $\pi_{w'}$. The expected values in (3-19) and (3-22) are equal.

Assume $\pi_w(s) = \pi_{w'}(s)$, $A^{\pi}(s,a)$ remains unchanged when w is changing, and sampling is sufficient, (3-23) can be estimated by

$$\nabla_w \hat{J} = \frac{1}{N_{(s_t,a_t)}} \sum_{(s_t,a_t)} \left[\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)} \hat{A}^{\pi}(s_t,a_t) \nabla_w log \pi_w(a_t|s_t)\right] \tag{3-24}$$

Where $N_{(s_t,a_t)}$ is the total number of (s_t,a_t) pairs of all Monte Carlo trajectories collected by the policy $\pi_{w'}$. With (3-24), multiple backpropagations using the same samples are possible. Since $\pi_w(a_t|s_t)\nabla_w log \pi_w(a_t|s_t) = \nabla_w \pi_w(a_t|s_t)$, the objective function is:

$$\hat{J} = \frac{1}{N_{(s_t,a_t)}} \sum_{(s_t,a_t)} \left[\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)} \hat{A}^{\pi}(s_t,a_t)\right] \tag{3-25}$$

We can simply input the objective function and samples (s_t,a_t) to a neural network optimizer and update the current policy π_w. However, experiments shows performance of π_w will increase and then decrease during the training process. This is because the assumptions of (3-24) are only tolerable if policy π_w and $\pi_{w'}$ are not very different.

Therefore, a reasonable compromise is stopping the optimization when the two policies become "too different", and then resample with the current policy π_w. How to measure the difference between two policies? We cannot simply compare the weights w and w' directly because similar weights can lead to very different decisions. The first method is the **Trust Region Policy Optimization** (TRPO) (Schulman, Levine, Abbeel, Jordan, & Moritz, 2015). It maximizes (3-25) with the constraint on the KL divergence:

$$KL(w, w') < \delta \tag{3-26}$$

The computation of KL divergence and optimization with an external constraint are complex. Therefore, another method called the **Proximal Policy Optimization** (PPO) uses the following objective function to limit the difference between π_w and $\pi_{w'}$ (Schulman, Wolski, Dhariwal, Radford, & Klimov, 2017):

$$\hat{J} = \frac{1}{N_{(s_t,a_t)}} \sum_{(s_t,a_t)} min\left\{\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)}\hat{A}^{\pi}(s_t,a_t), clip\left[\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right]\hat{A}^{\pi}(s_t,a_t)\right\} \tag{3-27}$$

$clip\left[\frac{\pi_w(a_t|s_t)}{\pi_{w'}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon\right]$ is shown in Figure 36.

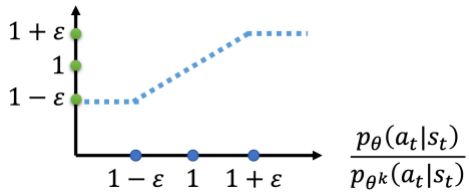


Figure 36: The Clip Function

PPO is much simpler than TRPO and can also ensure performance of π_w to increase. Both TRPO and PPO are seen as on-policy because π_w and $\pi_{w'}$ are not very different.

Other trust region methods include the **ACER** (Wang et al., 2016) and **ACKTR** (Wu, Mansimov, Grosse, Liao, & Ba, 2017). They are more complex and don't always have better performance than PPO.

3.4.3 Advantage Actor-Critic (A3C/A2C)

The A3C/A2C algorithms have both explicit policy function $\pi_w(a|s)$ and state-value function $\hat{V}(s)$. After a "worker" collects a complete trajectory using current policy π_w (called the **actor**), it estimates the advantage function $A^{\pi}(s,a)$ in (3-19) by

$$\hat{A}^{\pi}(s_t,a_t) = \hat{Q}^{\pi}(s_t,a_t) - \hat{V}^{\pi}(s_t) = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{N-1} r_{t+N-1} + \gamma^N \hat{V}_{\theta}^{\pi}(s_{t+N}) - \hat{V}_{\theta}^{\pi}(s_t) \tag{3-28}$$

(3-28) uses N-step TD sampling to estimate $Q_w^{\pi}(s_t,a_t)$. It's a better estimation compared with (3-21) because of low bias and variance of N-step TD. $\hat{V}_{\theta}^{\pi}(s_t)$ and $\hat{V}_{\theta}^{\pi}(s_{t+N})$ are computed using another neural network called the **critic**. Weights of the critic network are θ. Based on (3-28), the policy gradient is estimated by

$$\nabla_w \hat{J} = \frac{1}{T_n} \sum_{t=1}^{T_n} \left[\hat{A}^{\pi}(s_t,a_t) \nabla_w log \pi_w(a_t|s_t) + \varphi \nabla_w H\big(\pi_w(s_t)\big)\right] \tag{3-29}$$

Where $H\big(\pi_w(s_t)\big)$ is an entropy regularisation term and φ is the entropy coefficient. A policy with higher $H\big(\pi_w(s_t)\big)$ is more random and explore more states in training.

Such algorithms also need to update the critic network. The gradient is estimated by

$$\nabla_\theta Loss = \frac{1}{2T_n}\sum_{t=1}^{T_n}\nabla_\theta\left[\hat{Q}(s_t,a_t)-\hat{V}(s_t)\right]^2 \tag{3-30}$$

Here the loss function is an estimation of $\frac{1}{2}\mathbb{E}\left[\hat{Q}(s_t,a_t)-\hat{V}(s_t)\right]^2=0$. Note that the actor network and the critic network sometimes share some layers, as shown in Figure 37. In this case, we can combine the two loss functions together to compute a total gradient.

A3C/A2C improve the sampling efficiency by having n "workers" to sample at the same time. In **Asynchronous Advantage Actor-Critic** (A3C) algorithm, each "worker" repeats (Mnih et al., 2016)

Step 1: Copy current weights of the global actor network and critic network.
Step 2: Sample a complect trajectory in simulated environment.
Step 3: Calculate the policy gradient (3-29) and the value gradient (3-30)
Step 4: Update the global actor network once using the policy gradient (3-29)
Step 5: Update the global critic network once using the value gradient (3-30)

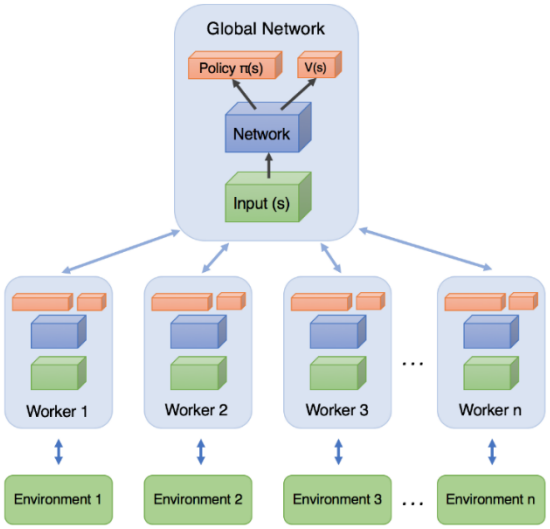


Figure 37: Architecture of A3C/A2C

In the **Advantage Actor-Critic** (A2C) algorithm, each worker will directly sends the sampled trajectory to the global computer. The global computer will calculate the total policy gradient and the total value gradient to update the networks. A2C is the synchronous version of A3C. It's simpler and generally has better performance.

A3C/A2C is a type of on-policy **actor-critic** algorithm (Konda, & Tsitsiklis, 2000). It loosely satisfies the i.i.d requirement in section 3.1.3. Samples from different workers are independent. However, for the actor network, distribution of $\hat{J}$ will be slightly changed by update of the critic network. On the other hand, distribution of the loss of the critic $\hat{V}^\pi(s)$ will be slightly changed by update of the actor network. Although such correlation is reduced by the N-step TD sampling, it still makes A3C/A2C a little difficult to train. Tricks to train A3C/A2C are like techniques for GAN (Pfau, & Vinyals, 2016). A3C/A2C can be combined with other policy gradient algorithms.

3.4.4 Algorithms for Continuous Action Space (DDPG/TD3/SAC)

All policy-based algorithms mentioned above can be used in continuous action space. In this case, input of a policy $\pi_w(a|s)$ is a state s, and output is parameters of probability distribution of an action a. Such $\pi_w(a|s)$ is a **stochastic policy**.

We can also adopt a **deterministic policy**, whose output is an action. Such policy is more efficient in continuous action space. An example is the **Deep Deterministic Policy Gradient** (DDPG) algorithm (Lillicrap et al., 2015).

DDPG is an off-policy actor-critic algorithm. As shown in Figure 38, it has a replay buffer and 4 neural networks: actor π_w, target π_{w-}, critic $\hat{Q}_\theta^\pi$, and target $\hat{Q}_{\theta-}^\pi$. The target actor π_{w-} and the target critic $\hat{Q}_{\theta-}^\pi$ are used to calculate the loss function of the current critic $\hat{Q}_\theta^\pi$.

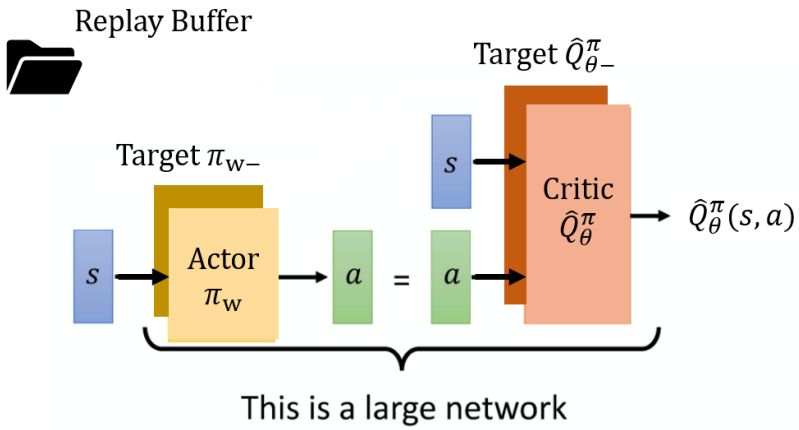


Figure 38: Components of DDPG

In each iteration, DDPG repeats:

Step 1: Take the action a_t, which is computed by π_w under current state s_t. We add Gaussian noise to output of π_w (e.g., a control torque) to encourage exploration, and slightly reduce the noise during training.

Step 2: Put an incomplete trajectory $(s_t, a_t, r_{t+1}, s_{t+1})$ to the replay buffer. $s_t \leftarrow s_{t+1}$.

Step 3: Randomly take N_τ incomplete trajectories τ from the buffer.

Step 4: Update the critic $\hat{Q}_\theta^\pi$ with the following gradient (gradient descent):

$$\nabla_\theta Loss = -\frac{1}{N_\tau}\sum_\tau \left[r_{t+1} + \gamma \hat{Q}_{\theta-}^\pi\left(s_{t+1}, \pi_{w-}(s_{t+1})\right) - \hat{Q}_\theta^\pi(s_t, a_t)\right]^2 \quad (3\text{-}31)$$

Step 5: Update the actor π_w with the following gradient (gradient ascent). This gradient can be computed by back propagation when weights of $\hat{Q}_\theta^\pi$ are fixed.

$$\nabla_w \hat{J} = \frac{1}{N_\tau}\sum_\tau \nabla_w \hat{Q}_\theta^\pi(s_t, a_t) \quad (3\text{-}32)$$

Step 6: Every C iteration, update the target actor π_{w-} and the target critic $\hat{Q}_{\theta-}^\pi$:

$$w^- \leftarrow \beta w + (1-\beta)w^-;\ \theta^- \leftarrow \beta\theta + (1-\beta)\theta^- \quad (3\text{-}33)$$

DDPG has some problems. Every C iteration, the distribution of (3-31) remains the same. However, the distribution of (3-32) is slightly changing with update of $\hat{Q}_\theta^\pi$. Only when update of $\hat{Q}_\theta^\pi$ stops, the distribution of (3-32) can be fixed.

At the same time, experience replay ensures selected samples are independent and improves the sampling efficiency. However, like DQN, DDPG tends to overestimate Q^{π}.

Twin Delayed DDPG (TD3) is an improvement on DDPG (Fujimoto, Hoof, & Meger, 2018). It learns two current critic networks $\hat{Q}^{\pi}_{\theta_1}$ and $\hat{Q}^{\pi}_{\theta_2}$ using the same gradient (3-34). It has 6 neural networks in total: 2 current critics $\hat{Q}^{\pi}_{\theta_1}$ and $\hat{Q}^{\pi}_{\theta_2}$, 2 target critics $\hat{Q}^{\pi}_{\theta_1^-}$ and $\hat{Q}^{\pi}_{\theta_2^-}$, a current actor π_w, and a target actor π_{w-}. TD3 adds 3 tricks to DDPG:

- **Clipped Double Q-Learning**. The target critic that gives smaller $\hat{Q}^{\pi}(s_t, a_t)$ will be used as the target in (3-34). Like the double DQN, it reduces overestimation of Q^{π}.

$$\nabla_{\theta_i} Loss = -\frac{1}{N_\tau}\sum_{\tau}\left[r_{t+1} + \gamma(1-d)\cdot min\hat{Q}^{\pi}_{\theta_i^-}(s_{t+1}, a_{TD3}(s_{t+1})) - \hat{Q}^{\pi}_{\theta_i}(s_t, a_t)\right]^2 \quad (3\text{-}34)$$

- **Delayed policy update**. TD3 updates the critic networks frequently and updates the actor network less frequently. It usually updates the critic twice and the actor once during the same time. In this way, TD3 reduces the influence of changing distribution of (3-32).
- **Target policy smoothing**. TD3 smooths $\hat{Q}^{\pi}(s_t, a_t)$ by

$$\begin{aligned} a_{TD3}(s_t) &= clip\left[\pi_w(s_t) + clip(\varepsilon, -c, c), a_{low}, a_{high}\right] for\ current\ actor \\ a_{TD3}(s_t) &= clip\left[\pi_{w-}(s_t) + clip(\varepsilon, -c, c), a_{low}, a_{high}\right] for\ target\ actor \end{aligned} \quad (3\text{-}35)$$

ε is a Gaussian noise. It makes the actor difficult to utilize the error of $\hat{Q}^{\pi}(s_t, a_t)$.

With these tricks, TD3 achieves much better performance than DDPG. However, TD3 has many hyperparameters to choose.

The **Soft Actor-Critic** (SAC) algorithm is also based on DDPG and has similar performance of TD3 (Haarnoja, Zhou, Abbeel, & Levine, 2018). But it has fewer hyperparameters. SAC doesn't maintain the target actor, so it has 5 neural networks in total. Other differences include:

- SAC maintains a stochastic policy, whose output is parameters of probability distribution of action a. SAC samples an action from the distribution and sends the action to the critic. It doesn't need explicit policy smoothing because noise from the stochastic distribution is sufficient.
- It adds an entropy term to the gradients of actor and critic.
- It computes the gradient of critic with the current actor since SAC doesn't have the target actor anymore.

3.5. Current Challenges and Progress of Reinforcement Learning

In this section we will briefly discuss main challenges of deep reinforcement learning (DRL) techniques. DRL is limited by current neural networks, which can only "fit some kinds of relations". To deal with such limitation, some researchers introduce new machine learning techniques (e.g., causal representation leaning), some other researchers hope to use some tricks (e.g., network structures and training methods) to "bypass" the limitation.

3.5.1 Low Sampling Efficiency

Low sampling efficiency means the agent needs too many samples to train. There are several tricks to solve this problem:

- Off-policy, such as the experience replay in DQN, DDPG, TD3 and SAC.
- Parallel sampling. Examples are A3C and A2C.
- Importance sampling. Examples are trust region methods TRPO/PPO/ACER/ACKTR.
- Model-based reinforcement learning.

We have discussed the first three tricks. Some large-scale distributed RL frameworks combine these tricks. For example, DeepMind's IMPALA (Espeholt et al., 2018) combines parallel sampling (like A2C) and importance sampling (like PPO). They also developed the APE-X (Horgan et al., 2018) which combines parallel sampling and experience replay.

All reinforcement mentioned above are **model-free**. By contrast, **model-based RL** learns a MDP model from interaction with the environment. The model usually includes the state transition probability $P(s'|s,a)$ and reward $R(s'|s,a)$. After that, it gets a policy by the probabilistic planning techniques in the chapter 2. The whole process is shown in Figure 39. Representatives of model-based RL include the prediction model (Silver, 2017), PLICO (Deisenroth & Rasmussen, 2011), and GPS (Levine & Koltun, 2013). A review of model-based RL can be found in (Moerland, Broekens, & Jonker, 2020).

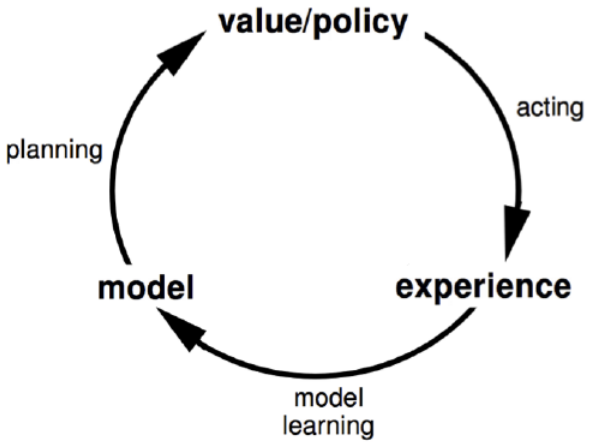


Figure 39: Process of Model-Based Reinforcement Learning

Model-based RL doesn't discard any samples. All samples are useful to fit the MDP model. By contrast, in model-free RL, only the samples collected by the current policy is useful and agents will throw outdated samples away (size of the replay buffer is limited). Therefore, it's not surprising that model-based RL has higher sampling efficiency.

Another advantage of model-based RL is better generalization. It can easily compute a policy for a new goal in the same MDP model. By contrast, model-free RL must be trained again.

Model-based RL algorithms also have some problems. They are more complex. They cannot solve tasks that cannot be modelled, such as natural language processing. At the same time, there is always error in the models they learn. Such error makes the policy not optimal. By contrast, model-free algorithms like Q-Learning are guaranteed to be optimal under some situations. As a result, model-based RL algorithms usually perform worse than model-free RL algorithms.

Some researchers try to combine model-free and model-based algorithms together. Such algorithms repeat:

Step 1: Collect a trajectory in real environment.

Step 2: Train the policy/value function using the trajectory.
Step 3: Train the MDP model using the trajectory.
Step 4: Collect a trajectory in the MDP model.
Step 5: Train the policy/value function using the trajectory from the MDP model.

Early works in this area include the Dyna (Sutton, 1991) and Dyna-2 (Silver, Sutton, & Müller, 2008) but the most well-known one is DeepMind's MuZero (Schrittwieser et al., 2020). As DeepMind's latest game agent, it performs better, needs fewer samples, and even doesn't need to know rules of the game. MuZero can be used in both discrete and continuous action space. We will briefly discuss it in section 3.6. A latest improvement on MuZero is EfficientZero (Ye, Liu, Kurutach, Abbeel, & Gao, 2021), which can even master the Atari game in a shorter time than human beings.

3.5.2 Sparse Reward

Rewards in some tasks are sparse, which means agents seldomly get a reward. Training becomes difficult in this case. There are several ways to alleviate this problem.

Better exploration policies may be helpful. We have introduced a simple ε-greedy policy, which randomly takes actions with probability ε. Similar policies include the Boltzmann policy (Luce, 2012), Thompson Sampling (Chapelle & Li, 2011), **Upper Confidence Bound UCB** (Auer, Cesa-Bianchi, & Fischer, 2002). For example, the UCB1 policy always chooses an action that maximizes

$$U(s,a) = \hat{Q}(s,a) + \sqrt{\frac{2lnN(s)}{lnN(s,a)}} \tag{3-36}$$

where $N(s,a)$ is the number of time that the agent tries action a in state s, and $N(s)$ is the number of time that the agent is in state s. Given some consumptions, we can prove that as number of calls → ∞, UCB's choices become optimal. Upper Confidence Bounds for Tree UCT (Kocsis & Szepesvári, 2006) is a variant in the Monte Carlo Tree Search (MTCS). The MCTS algorithm with UCT is powerful in board games. It beat professional Go players in 2006 for the first time and were still the basic algorithm of AlphaGo in 2016 (Silver et al., 2016).

We can also encourage exploration by adding **intrinsic rewards**. Such rewards can be given according to the number of visits to the current state (**count-based exploration**). An example is the **hash-based counts** (Tang et al., 2017), which adds visitation count by an auto-encoder. On the other hand, the agent can use a neural network to predict the next state. If the real next state is far from the prediction, the agent can get a reward because it finds something new (**curiosity-based exploration**). Examples are the ICM (Pathak, Agrawal, Efros, & Darrell, 2017) and RND (Burda, Edwards, Storkey, & Klimov, 2018) methods.

Another approach to deal with sparse reward is **hierarchical reinforcement learning** (HRL), which is like the HTN in Chapter 2. In HRL, high-level agents set goals of low-level agents. For example, the H-DQN algorithm uses a 2-level DQN to acquire higher scores than the original DQN in the Atari game (Kulkarni, Narasimhan, Saeedi, & Tenenbaum, 2016). A survey of HRL can be found in (Pateria, Subagdja, Tan, & Quek, 2021).

Reward shaping and **curriculum learning** are problem-specific tricks. In reward shaping, researchers carefully design rewards in every step to guide the agent. In curriculum learning, the agent first learns how to perform simple tasks, and then learn how to difficult tasks. These 2 tricks are shown in Figure 40.

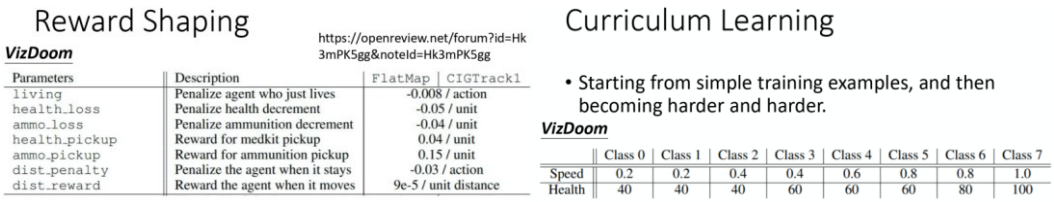


Figure 40: Reward Shaping and Curriculum Learning

3.5.3 Imitation Learning

In many tasks, it's difficult to define rewards. For example, it takes too much time to carefully define every reward in an autonomous driving task. Reinforcement learning algorithms cannot be used without rewards. However, we can still use imitation learning (IL), which learns from demonstration of experts. IL and RL share many fundamental technologies.

The simplest IL is **behaviour cloning**. Experts provide demonstration samples like $(s_1, a_1), (s_2, a_2), \dots (s_n, a_n)$. Then we train a policy network whose input is a state s and output is an action a. In fact, behaviour cloning is supervised learning. However, a behaviour cloning agent cannot deal with a situation that has never been demonstrated.

Another approach is the **inverse reinforcement learning** (IRL). The IRL agent will learn how to set rewards from demonstration of experts. After that, we can train the agent with the rewards and RL algorithms. A famous example is the Generative Adversarial Imitation Learning GAIL (Ho & Ermon, 2016), which takes idea from GAN. In GAIL, the discriminator tries to assign high rewards to the expert trajectories, and low rewards to trajectories from the generator. At the same time, the generator learns to maximize its rewards. A survey of imitation learning is (Hussein, Gaber, Elyan, & Jayne, 2017).

For traditional reinforcement tasks that we can define rewards, IL may also be helpful. For example, in section 3.3.3 we introduce DQN from demonstration (Hester et al., 2018), which adds expert trajectories to the replay buffer to accelerate learning. AlphaGo also used expert experience to get a good initialization (Silver et al., 2016).

3.5.4 Migration to Other Problems

An RL model is trained for a specific problem. When the problem or the goal is slightly changed, the trained model becomes useless. There are several approaches to alleviate the problem.

The first approach is **multi-task learning**. The initial agent is trained on some supporting problems. After that, we continue to train this initial agent on a target problem to fine-tune the parameters. The second step usually takes much less time and sampling. The idea is like the pre-training in natural language processing.

We can also use the goal as an input to the policy network or the value network. In this case, the agent can be applied to the same problem with different goals. However, training will become more difficult, and capability of the neural networks should be strong enough. Examples are UVFA (Schaul, Horgan, Gregor, & Silver, 2015), UNREAL (Jaderberg et al., 2016), and HER (Andrychowicz et al., 2017)

Another approach is **meta-learning**. Its input includes the training set of the target problem, and output can be parameters, structures, activation functions, or weight update methods. In short, it learns how to design an RL agent. For example, the MAML algorithm learns how to set initial weights of neural networks for different problems (Finn, Abbeel, & Levine, 2017).

Reinforcement learning can be used for causal reasoning (Zhu, Ng & Chen, 2019). On the other hand, **causal reasoning** can also help RL (Dasgupta et al., 2019). Reversibility is another issue in causal representation based RL (Grinsztajn, Ferret, Pietquin, & Preux, 2021).

A survey of generalization of RL is (Kirk, Zhang, Grefenstette, & Rocktäschel, 2021).

3.5.5 Other Topics

Like the planning area, **multi-agent** is also an important topic in reinforcement learning. Interesting research (Siu et al., 2021) shows that RL agents are not your good team members at this moment.

Safety is another problem of current RL, especially model-free RL. Many classical algorithms like DQN cannot withstand adversarial attacks (Behzadan & Munir, 2017).

At this moment, most of RL agents can only handle single type of input. However, in practice there can be multiple types of inputs: text, image, sound… As an example, DeepMind's AlphaStar can deal with **multimodal inputs**. We will discuss it in the next sections.

Because of low sampling efficiency, most of successful RL are trained in simulators. However, there is usually a gap between simulation and realistic. As shown by (Rajeswaran, Lowrey, Todorov, & Kakade, 2017), some RL experiments show good performance because their simulators are too simple. To be honest, some experiments cannot even be repeated at all (Islam, Henderson, Gomrokchi, & Precup, 2017). This situation will be changed if good simulators fill the **reality gap**. Digital twins supported by neural networks may be useful here.

3.6. Applications

At this stage, reinforcement learning is most mature in games. It's easy to collect large amount of samples in games and the reward is clear (higher score in a game). The most famous work is the game AI of DeepMind. In this section, we will briefly discuss:

- AlphaGo Zero, a model-free RL agent in a fully observable environment. Humans write rules of the game in the agent.
- MuZero, a model-based RL agent in a fully observable environment. It learns rules from the environment.
- AlphaStar, an RL agent in partially observable and dynamic environment.

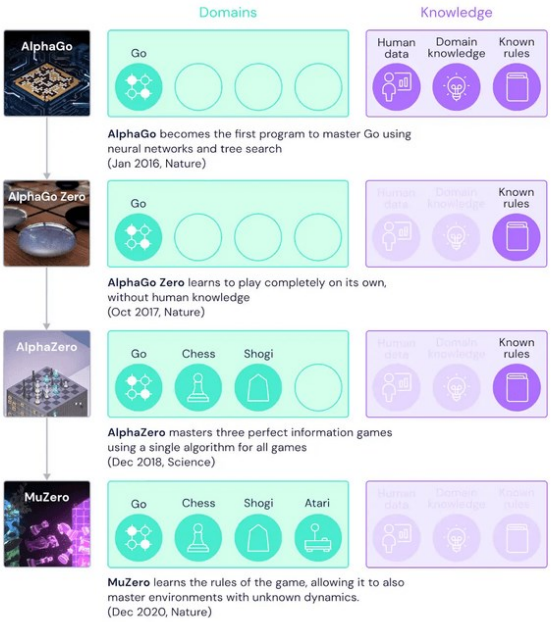


Figure 41: Some Game AI of DeepMind

3.6.1 AlphaGo Zero (DeepMind, 2017)

AlphaGo Zero relies on the Monte Carlo Tree Search (MCTS). Before taking an action, the agent builds a MCTS and uses the final data in the tree to make the decision. At each node s' of the tree, there is a value estimation $\hat{Q}(s,a)$ and a prior estimation $\hat{P}(a|s)$, and s is the father node, as shown by Figure 42. For simplicity, we assume an action only has a single outcome.

An MCTS is constructed by repeating:

- Step 1: **Selection** always starts at the root of the tree, i.e., the current state in the game. At each node, it uses a scoring function $U(s,a)$ to compare different actions a and choose the largest one. Computation of $U(s,a)$ is slightly different from (3-36) in UCB1:

$$U(s,a) = \hat{Q}(s,a) + \hat{P}(a|s)\frac{c\sqrt{lnN(s)}}{1+lnN(s,a)} \tag{3-37}$$

- Step 2: **Expansion**. The selection process stops at a leaf node. All children of this leaf node will be expanded according to rules of the game. These new nodes contain $\hat{Q}(s,a)$ and $\hat{P}(a|s)$ which are initialized to 0.

- Step 3: **Simulation**. For each new node, the agent will compute $\hat{P}(a|s)$ according to some rules. And then do a simulation from this node to the end of the game. In the simulation, the agent will choose actions according to some simple policy, e.g., fast rollout. The agent gets a result of this game after the simulation.

- Step 4: **Backpropagation**. Use the result of the simulation to update $\hat{Q}(s,a)$ in the path.

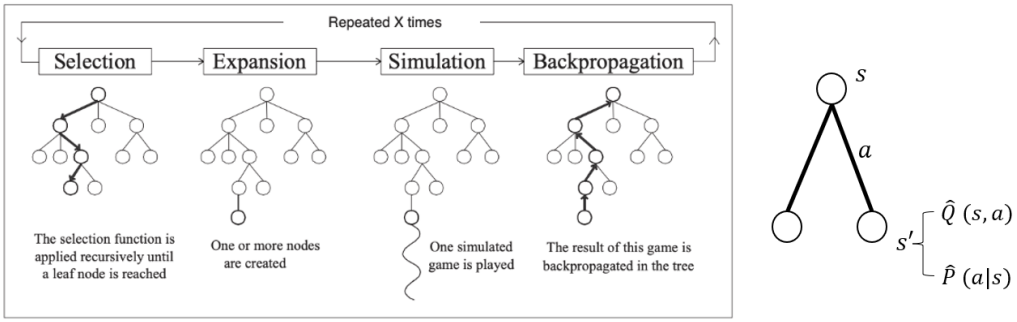


Figure 42: Outline of a Monte Carlo Search

The MCTS algorithm with UCB (Kocsis & Szepesvári, 2006) significantly improved performance of game AI. However, the step 3 simulation still takes too much time. AlphaGo and AlphaGo Zero (Silver et al., 2017) used a convolution neural network (ResNet) to accelerate the simulation. The input of the neural network includes previous 8 states of the Go board. The output includes $\hat{P}(a|s)$ of the new node and change to $\hat{Q}(s,a)$ of all nodes on the path.

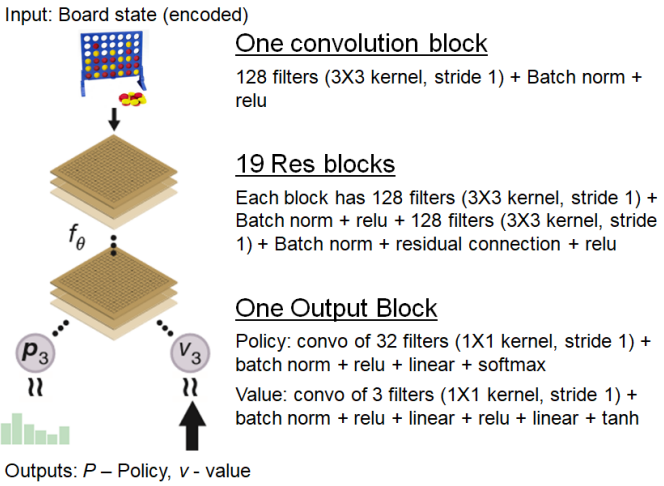


Figure 43: Neural Network Architecture of AlphaGo Zero

After around 1600 MCTS, AlphaGo Zero chooses the action in the game to maximize the following formula:

$$\pi(a|s_t) = \frac{N(s_t,a)^{1/\tau}}{N(s_t)^{1/\tau}} \tag{3-38}$$

Where s_t is the root node of the tree at step t of the game. τ is a parameter to control the degree of exploration. $\tau = 0$ means greedy action selection, and $\tau = inf$ means sampling actions uniformly.

In the training process, AlphaGo Zero will play with itself to generate samples. The loss function of the neural network is simply (excluding regularisation terms):

$$Loss = \sum_t \left\{ \left[Q(s_t,a_r) - \hat{Q}(s_t,a_t) \right]^2 - \sum_a \pi(a|s_t) \cdot ln\left(\hat{P}(a|s_t)\right) \right\} \tag{3-39}$$

3.6.2 MuZero (DeepMind, 2020)

AlphaGo Zero and its variant AlphaZero achieve good performance in board games like go, chess, and shogi. However, in step 2 “expansion”, MCTS still needs rules of the game to find successor states, i.e., children of a node. By contrast, the MuZero (Schrittwieser et al., 2020) automatically learns rules of the game and builds a MDP model.

There are 3 neural networks in MuZero:

- The **representation** network h maps from a set of observations to a hidden state s_t.
- The **dynamics** network g maps from a state s_t to the next state s_{t+1} based on an action a_t. It also estimates the reward r_{t+1} observed in this transition.
- the **prediction** network f is like the neural network in AlphaGo Zero.

MuZero uses networks g and h to find successor hidden states in step 2 “expansion”. The overall training process is like the one in section 3.5.1, as shown by Figure 44.

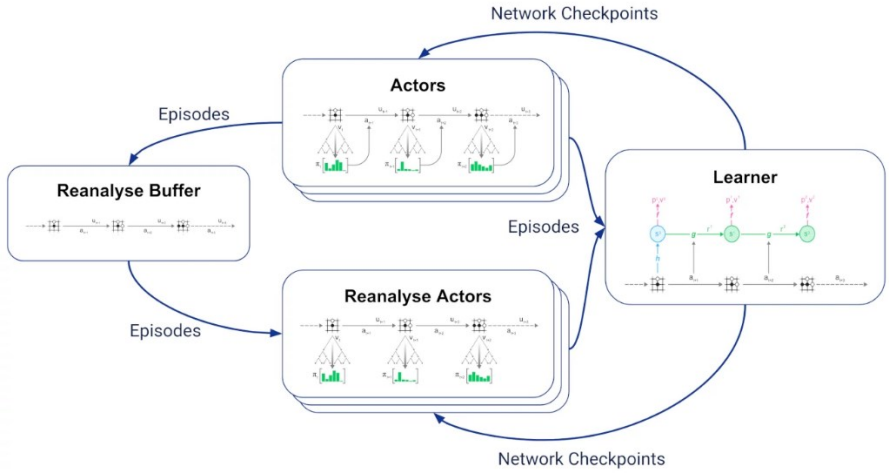


Figure 44: Parallel Training of MuZero

MuZero sets a new state of the art for reinforcement learning algorithms, outperforming all prior algorithms on the Atari game, as shown by Figure 45. It also masters more than 50 other games.

Agent	Median	Mean	Env. Frames
Ape-X	434.1%	1695.6%	22.8B
R2D2	1920.6%	4024.9%	37.5B
MuZero	**2041.1%**	**4999.2%**	20.0B
IMPALA	191.8%	957.6%	200M
Rainbow	231.1%	-	200M
UNREAL	250%	880%	200M
LASER	431%	-	200M
MuZero Reanalyse	**731.1%**	**2168.9%**	200M

Performance on the Atari suite using either 200M or 20B frames per training run. MuZero achieves a new state of the art in both settings. All scores are normalised to the performance of human testers (100%), with the best results for each setting highlighted in bold.

Figure 45: Performance of RL Algorithms in the Atari Game

3.6.3 AlphaStar (DeepMind, 2019)

The games played by AlphaGo Zero and MuZero are relatively simple: the state is fully observable, the input only includes images of the screen, and the environment is relatively static. By contrast, DeepMind’s AlphaStar agent is designed for the StarCraft II game, which is partially observable,

dynamic and has multimodal inputs (Vinyals et al., 2019). The architecture of AlphaStar is shown in Figure 46.

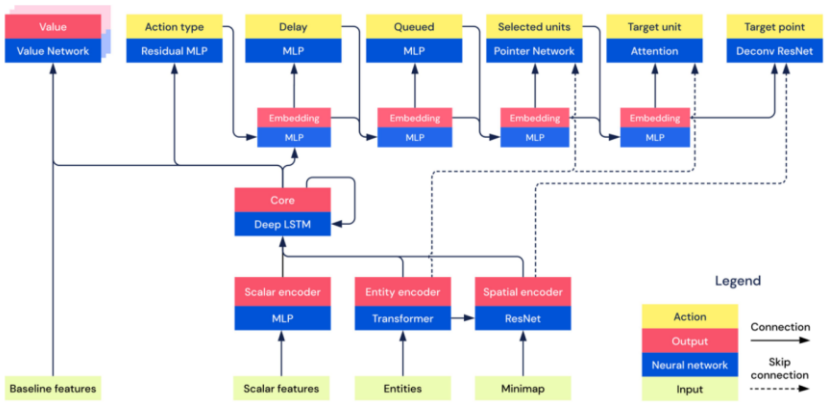


Figure 46: Architecture of AlphaStar

Limited by pages, it's difficult to introduce details of AlphaStar here. But there are some important ideas to mention:

- AlphaStar allows multimodal inputs and provides capability like temporal planning.
- AlphaStar combines reinforcement learning and imitation learning.
- The structure and training strategies of the neural networks are complex.
- It takes significant amount of computation resource.

4 The Delfi-PQ Mission

PocketQube is a new form factor of very small satellites. The size of 1P PocketQube is around 5×5×5cm, smaller than 10×10×10cm of 1U CubeSat. Since they are smaller, PocketQubes have stronger constraints on mass, size, power, and communication budget. However, for some applications like education, technology demonstration, gravity / magnetic / radiation multi-point measurement, PocketQubes are cheap and competitive (Bouwmeester et al., 2020).

Delfi-PQ is a 3P PocketQube and third student satellite made in TU Delft (Radu et.al, 2018). The purpose of the mission includes education and technology demonstration. In this chapter, we will briefly discuss the hardware and software design of the satellite. We will also discuss how the planning and learning technology can help this mission.

4.1. Overview of Subsystems

The first Delfi-PQ has 7 subsystems, the On-Board Computer (OBC), the Communication System (COMMS), the Antenna Deployment Board (ADB), the Electrical Power System (EPS), the Attitude Determination and Control System (ADCS), a low frequency radio payload (LOBE-P), and a redundant on-board computer. An Electrical Ground Support Equipment (EGSE) board is used in debugging.

Each subsystem has a Texas Instrument MSP432P4111 microcontroller, which controls how the subsystem works. All these microcontrollers are connected to an RS-485 bus with speed of 115.2kbps. The OBC is the master of the bus. Only the OBC can actively send frames over the bus and other subsystems only reply passively. The bus only allows half-duplex communication, and each frame has up to 256 bytes. Each microcontroller also has JTAG pins, which can be connected to a PC via a JLINK connector.

The 48MHz microcontroller has 2MB Flash and 256KB SRAM. There is also a 512KB FRAM for each microcontroller. Information in a FRAM will not be lost after a reset. However, only the OBC has a 2GB SD-card to store telemetry.

Every microcontroller should kick an external watchdog on the board at least once every 2.5 second, otherwise the board will be reset. At the same time, a microcontroller should kick an internal watchdog at least once every 178 second, otherwise the controller will be reset. These are basic measures to deal with space radiation.

The EPS consists of the battery board (1500mAh, 2 batteries of 3.7V), the main EPS board and solar panels. EPS manages 4 power lines, and each line has some subsystems on it. Depending on commands from the OBC, EPS can enable, disable, reset, or power cycle a power line. EPS is designed to be always running after the Delfi-PQ is released from the deployer. Therefore, a reset of the EPS leads to a reset of the whole satellite. If the battery voltage is lower than 3.6V, the OBC should commend the EPS to disable the power lines of unnecessary subsystems.

The COMMS receives and decodes signal from the ground station. It automatically puts the ground commands into the RX queue. If the OBC requests ground commands from the queue, COMMS will take a ground command from the queue and send it to the OBC. On the other hand, if the OBC needs to send a message to the ground station, it will command COMMS to put the message to the TX queue. COMMS will automatically sends all messages in TX queue to the ground. The RX/TX queue can store up to ~200 messages. The communication is full-duplex with a nominal speed of

1200 bps, or a higher speed of 9600 bps at 2W power consumption. In emergent case, if the COMMS receives a special commend from the ground, it will raise a special line to reset the EPS, which will reset the whole satellite.

Another important subsystem is the ADCS. It has a BOSCH BMX055 sensor chip including a gyroscope, an accelerometer, and a geomagnetic sensor. It also has 3 house-made coils as magnetorquers to control the rotational speed of the satellite. If the rotation speed is larger than 5 deg/second, the OBC will command the ADCS to slow down the rotation.

The OBC controls how the subsystems work. It has a state machine which cover very basic operations of the satellite. As shown by Figure 47, the state machine has five modes: initial mode, antenna deployment mode, safe mode, ADCS mode, and normal operation mode. In the normal operations mode, the OBC will request telemetry from every subsystem periodically, save the telemetry is in its SD card and send it to the ground via COMMS. OBC will also periodically request ground commands from the COMMS. If the command is for OBC itself, it will deal with it and reply to the ground station. If the command is for another subsystem, OBC will forward the command to that subsystem, wait for the reply, and send reply to the ground.

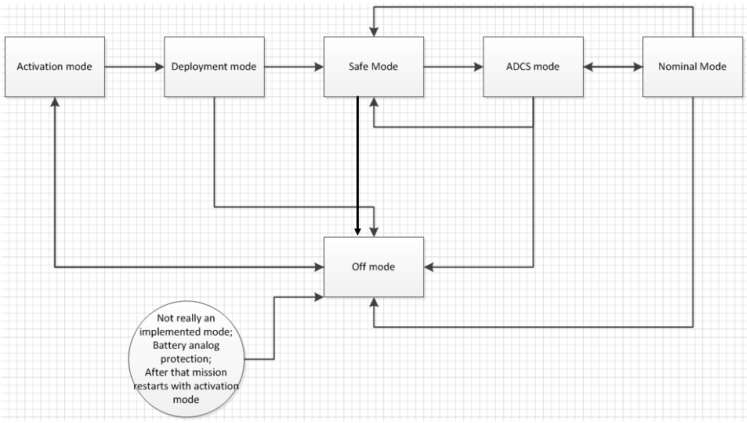


Figure 47: The State Machine in OBC

ADB is used to deploy the antennas after the satellite is released from the deployer. The payload is another radio which will generate scientific data. The redundant on-board computer board only has a MSP432 and some basic components.

The EGSE board can be put on the bus during debugging. It has a MSP432 which transfers frames between the bus and a USB connector, which is connected to a PC. On the PC, frames are converted to the JSON format and be sent via Internet. In this way, remote testing can be achieved. EGSE can listen to every frame over the bus and be the master of the bus.

4.2. Onboard Software

The onboard software can be seen to have 3 parts:

- **Drivers**, i.e., driver functions of peripherals.
- **DelfiPQcore**, a lightweight operating system with some basic helper functions.
- **Application**. Different microcontrollers run different applications on the same drivers and the DelfiPQcore.

We will discuss the general workflow of the onboard software in section 4.2.1, basic concepts of DelfiPQcore in section 4.2.2, and applications in section 4.2.3. Most of information of this section are taken from internal reports.

4.2.1 Workflow of Onboard Software

The general workflow of the programs can be described as a sequence of initialization steps, after which the program will go into a continuous task loop, as shown by Figure 48. In this loop it does:

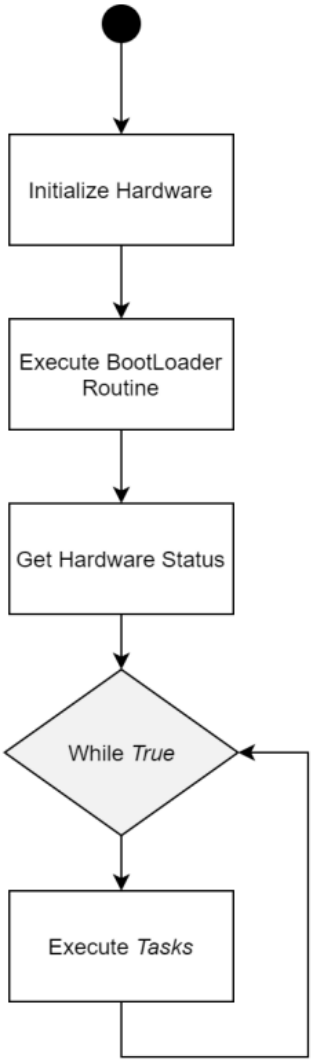


Figure 48: Workflow of Onboard Software of Delfi-PQ

Step 1: Initialize Hardware

This step should initialize the critical hardware components of the module. This ranges from general hardware in the MCU, such as initializing the clocks and busses, to specific hardware for this module, such as current sensors.

Step 2: Execute Bootloader Routine

One of the core features of the software is the possibility to load different software versions from the flash memory of the MCU.

Step 3: Get Hardware Status

As soon as possible in the startup routine, critical hardware status indicators should be collected and stored. These critical status indicators include the reset status (the reason for last reboot) and

possible clock faults.

Step 4: Task Execution Routine

After the operating system has completed its boot steps it starts a continuous task execution routine, which can be considered a simple non-preemptive, non-prioritized, linear scheduler (round-robin).

4.2.2 Important Concepts of Delfi-OS

In this section we discuss some basic concepts and functions in the operating system.

Task

After the operating system is started, tasks can be executed. Any processing/data collection or any other action executed by the device, is preferably described as a 'Task'. A task consists of an Initializer Function, a function executed once during the initialization of the Scheduler (Task Manager), and a User Function, the function executed every Iteration of the Task. Lastly the task has an Execution Flag, raising this flag will tell the Task Manager that this Task is ready for execution. If the Execution Flag is not raised, the Task will not be executed and 'skipped' by the Task Manager. The Execution Flag can be raised either externally by another task or using any interrupt routine, this action will henceforth be called 'notifying' a Task.

PeriodicTask

Some tasks will require periodic execution, and there might not be any clear external trigger available to notify such tasks (such as a Telemetry collection Task). Such a Task can be defined as a PeriodicTask, these tasks include another parameter which contains the required amount of 'counts' for the task to be notified (currently, 1 'count' is approximately 0.1 second). An external object, the TaskNotifier, will notify the period tasks assigned to it using an interrupt routine.

Service

The most common source of notifying a task is from an external trigger over the satellite bus. The satellite bus driver will receive bytes over the bus using a hardware-interrupt routine, if a full frame is received, a CommandHandler Task is notified and the received frame is copied into its buffer. The scheduler will consecutively execute the CommandHandler Task since the Execution Flag is raised. The CommandHandler will read the DataFrame and will 'poll' so called Services it has registered to it. When a Service detects that the received frame is intended for this service, it will process the received frame, set a response frame, and notify the CommandHandler that it has processed this frame. The CommandHandler will then stop polling other services and reply over the bus. A user should create a service for every functionality that is required over the satellite bus.

PQ9Bus and PQ9Frame

Though the commandHandler is built to handle any frame it is supplied, the services used in the Delfi-PQ project are frame-specific. In this project, the frame used is a so-called PQ9Frame as is built in the following way:

Byte number	Description
0	Destination Address
1	Payload Size
2	Source Address
3 … N	Payload
N + 1	CRC1
N + 2	CRC2

Figure 49: PQ9Frame Definition

Whereas the frame Payload is described as follows:

Byte number	Description
0	Service Number
1	Action: - 0: ERROR - 1: REQUEST - 2: REPLY
2 … N	Service Payload

Figure 50: PQ9Frame Payload Definition

Bootloader and OTA Update

As mentioned earlier one of the core functionalities of the OS is to execute a different software version from flash. This is handled by the so-called BootLoader. This bootloader requires an external memory (FRAM) to be present that holds (non-volatile) information regarding which memory-slot needs to be executed, whether the last execution was successful and how-many unintended reboots happened while in a certain executed slot. If this information tells the bootloader that the target slot is broken or has issues (or if the external FRAM is unavailable), it will fall back on the default slot (Slot 0). The device has three slots available, Slot 0, the default slot that is protected in flash and cannot be reprogrammed, and Slot 1 & Slot 2, which are allowed to be reprogrammed. Using this functionality an OTA (Over-The-Air) update can be executed in situ. Currently a service is implemented (SoftwareUpdateService), that allows a binary file transfer of a new software version over the bus to be programmed in flash. Using this, a module can be reprogrammed externally and even in orbit. Note that for this functionality to work the FRAM needs to be present.

4.2.3 Tasks and Services in Each Subsystem

Tasks shared by every subsystem:

- **Timer task**: A periodic task which collects telemetry of the subsystem every second.
- **CommandHandler**: It processes received frames and replies. CommandHandler is raised when a frame comes.

Service shared by every subsystem:

- **Ping service**: It replies to the ping command.
- **Reset service**: A microcontroller can be reset, or power cycled by a command.
- **FRAM service**: FRAM can be read, written, or erased by a command from the bus.
- **Housekeeping service**: Send the telemetry collected by the Timer task as a response.
- **Software update service**: Handle commands for OTA. New binary software is the payload of some OTA commands.

Special tasks of ADB:

- A function in the burn service is set as a periodic task.

Special services of ADB:

- **Burn service**: It burns the wire that locks the antenna, so the antenna is deployed.

Special tasks of ADCS:

- None. Readout of attitude sensors is part of the telemetry, and the magnetorquer can be controlled via the coil service.

Special services of ADCS:

- **Coil service**: Set states of the magnetorquers.

Special tasks of COMMS:

- **CommRadio**: This task is raised by the Radio service.

Special services of COMMS:

- **Radio service**: It includes a set of functions to interact with COMMS.

Special tasks of EPS:

- None.

Special services of EPS:

- **Power bus handler**: Set states of the power lines.

Special tasks of LOBE-P (payload):

- **lobepRadio**: like CommRadio, but working on lower frequency

Special services of LOBE-P (payload):

- **lobep service**: like the radio service, but working on lower frequency

Special tasks of OBC:

- **State Machine**: It's a periodic task that runs the simple state machine in section 4.1.
- **File system**: This task is raised by the telemetry request service to retrieve telemetry files from the SD card asynchronously or raised by the state machine to store telemetry in the SD card.

Special services of OBC:

- **State machine service**: get / set the current state; enable beacon or reset the state machine.
- **Telemetry request service**: request telemetry file from the SD card or format the SD card.
- **Bootloader override service**: command the microcontroller to jump to a specific slot. It should be a service shared by multiple subsystems but is only an OBC service at this moment.

4.3. Apply Autonomy in the Mission

As mentioned in Chapter 1, three possible applications of planning/learning in the Delfi-PQ mission are identified:

App1: Autonomous operation of the satellite.
App2: Autonomous recovery from faults.
App3: Automated testing of onboard software.

For onboard autonomous operation or fault recovery, the computing resource is too limited, e.g., 48MHz microcontrollers with 256KB SRAM. Other missions with onboard planners have stronger computing capability and more mature operating system support:

- The early Deep Space 1 mission has a 33MHz RAD6000 CPU running the VxWorks OS with 96MB hardened RAM (Williams et al., 1997).
- The IPEX 1U CubeSat has a 400MHz ARM9 CPU running the Linux operating system with 128MB RAM (Chien et al., 2017).
- The ASTERIA 6U CubeSat has a CORTEX 160 flight computer, which has a 400MHz PowerPC 405 CPU running the Linux with 64MB DSRAM (Smith et al., 2018). The MEXEC planner needs to fit in 2MB of memory, so the engineers put a lot of effort into it (Troesch et al., 2020).

We can operate Delfi-PQ on the ground since there are more computing resources. However, the operation is simple so planning/learning techniques may not be necessary. Basically, we only need to decide when to open the payload or when to downlink. In this case, a greedy scheduling algorithm like (Rabideau, Chien, Galer, Nespoli, & Costa, 2017) is enough. Operation of other PocketQubes is also relatively simple because of limited performance.

The situation is similar for fault recovery on the ground. There are engineers on the ground to deal with faults, so Delfi-PQ doesn't need complex ground software to do the same thing. On the other hand, automated telemetry processing is helpful but out of the scope of this document.

Automated testing of onboard software seems to be doable and helpful:

- It's performed on the ground so not limited by onboard computing resources.
- Although planning/learning techniques are not commonly used in space industry, they have been used in software engineering.
- There are many "actions" to choose in a test, so test case generation is complex enough to use the planning/learning techniques.
- Currently we write all test cases by hand, which takes a lot of effort and cannot cover every edge case. It will be helpful if we can automate this process.
- It's also helpful to other PocketQubes because their software is updated frequently.

In the next chapter, we will discuss how to apply such techniques in the mission and compare them with the existing methods.

5 Apply Autonomy in Testing

This chapter proposes to use planning and reinforcement learning techniques to test onboard software of Delfi-PQ:

- It introduces software testing in general in section 5.1.
- After that, it reviews traditional test case generation methods in section 5.2.
- Section 5.3 briefly reviews the literatures to apply planning in software testing.
- More and more researchers and companies are using reinforcement learning to test software in recent years. Section 5.4 reviews the progress in this field.
- In section 5.5, we will discuss which gap can be filled in this area, and how much it is related to the Delfi-PQ mission.
- Research questions are proposed in section 5.6.

5.1. A Bite on Software Correctness Testing

Generally, software testing has the following steps:

- Step 1: Analyse the requirement document.
- Step 2: Make a test plan, including testing methods, environment settings, etc.
- Step 3: Set the testing environment. For embedded software, tests can be run on target boards, in a hardware-in-the-loop environment, or in a fully virtual machine.
- Step 4: Write test cases, execute them, and detect faults. A **test case** is a set of actions executed to test a system. A **test suite** includes multiple test cases.
- Step 5: Record the faults and let software engineers to debug.

Although step 3 is also important for embedded software testing, we focus on step 4 in this chapter because it can be seen as a sequential decision-making problem. Also note that we mainly discuss correctness tests, instead of performance tests, safety tests, portability tests, etc. A comprehensive introduction to software testing is (Ammann & Offutt, 2016).

Methods and levels of testing are briefly introduced in the following subsections.

5.1.1 Testing Methods

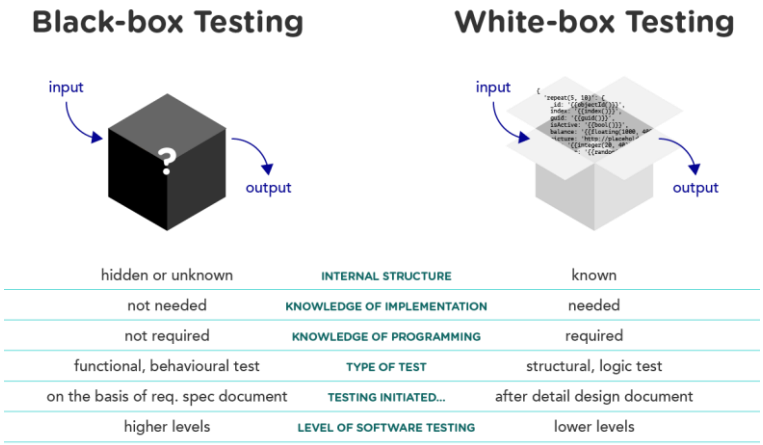


Figure 50: Black Box and White Box Testing

Software testing methods can be split into 2 categories, “**black box**” and “**white box**” methods. The black box testing is usually performed at a high level, where the testers do not need to know the system in detail. On the other hand, the white box testing needs details of the system, such as behaviours of a component or source code (Palmieri, 2013). The white box testing is more precise, but the creation and execution of test cases will take more effort.

Typical black box testing methods include:

- **Compare output of software with expected values**. The expected output can be store in a decision table, or a state transition table, etc.
- **All-pair testing / pairwise testing**. It tries to minimize the number of test cases while ensuring that all combinations of inputs are tested. This method assumes effects of different input variables are independent. An example is shown in Figure 51.

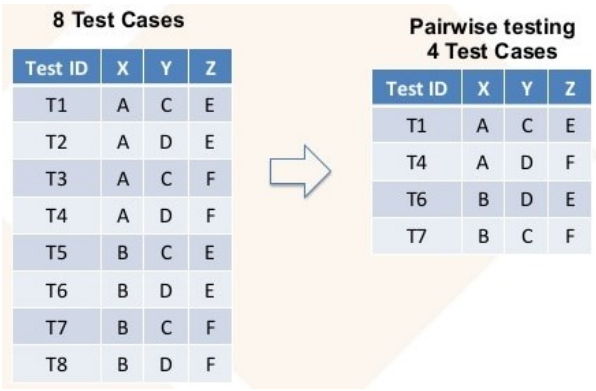


Figure 51: An Example of Pairwise Testing

- **Equivalence partitioning** also reduces number of test cases. It divides the input in several equivalent partition. Input values from the same partition have the same effect. Therefore, a few samples from an equivalent partition can represent other samples in the same partition. Figure 52 shows an example of equivalence partitioning.

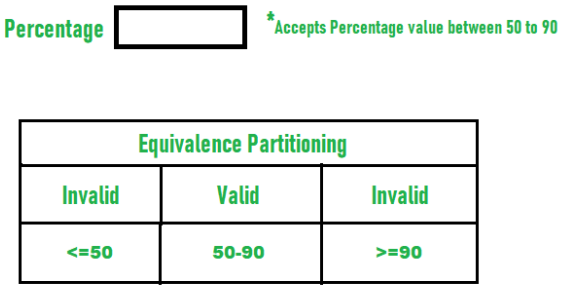


Figure 52: An Example of Equivalence Partitioning

- **Boundary analysis** is an improvement on equivalence partitioning, which only samples input values around the boundaries, as shown in Figure 53.

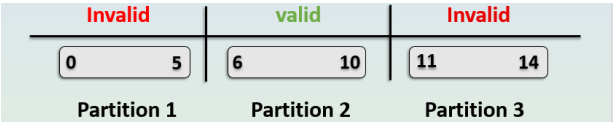


Figure 53: An Example of Boundary Analysis

Typical white box testing methods include:

- **Basis path testing** firstly constructs a control flow graph of the tested software, as shown in Figure 54. After that, it looks for all independent paths and executes them.

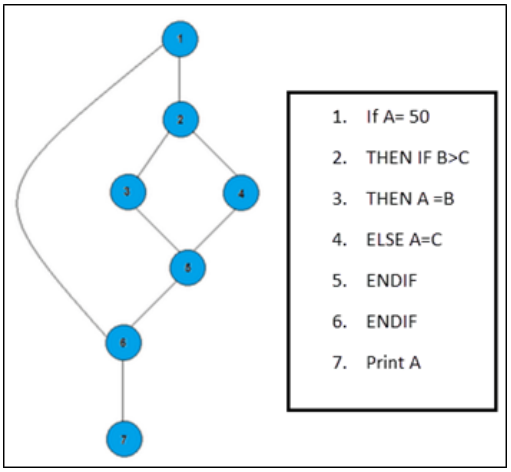


Figure 54: Control Flow Graph of a Simple Program

- **Control structure testing** is a set of improvements on the basis path testing. For example, **conditioning testing** looks for paths that cover all possible conditions. For each variable in a program, **dataflow testing** covers at least a path from definition of the variable to utilization of the variable.

- **Mutation testing** inserts faults in the software to check whether current test cases can find the faults. It can also be used to test the fault handling mechanism.

There are still many other testing methods that are not included in the subsection.

5.1.2 Testing Levels

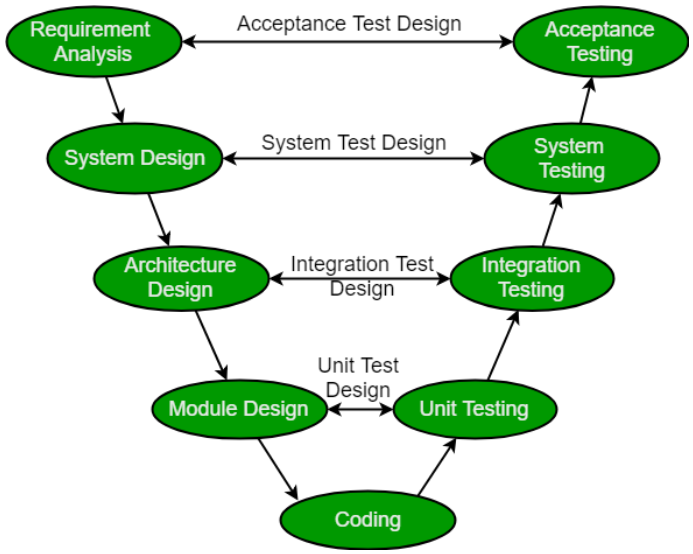


Figure 55: The V-Model of Software Development

The V-Model is system engineering is also used in software development. As shown in Figure 55, it usually has **unit testing**, **integration testing**, **system testing** and **acceptance testing**. For higher level tests, the system becomes more complex, and it's usually more difficult for testers to access internal information of the system.

Before coding, we may also need to **test requirements** and system design, which can be done by technologies like model checking. After the acceptance test, if the software is updated and has some new capabilities, it needs **regression tests** to ensure the original capabilities are not affected. We can reuse test cases in regression tests.

5.1.3 Test Embedded Software

Testing embedded software is different from testing conventional software (e.g., PC, web or mobile applications). The main challenges include:

- Embedded software run on a target board with limited computing resources.
- It's more difficult to get debug information from the target board, especially when the microcontrollers don't support some debug capabilities, like tracing. This raises a need for sophisticated instrumentation and probing when testing embedded systems.
- Apart from commands from a host PC, a target board may also sense signals like temperature, acceleration, light intensity, wireless communication, etc. These signals will affect the test, and sometimes testers need to provide such signals during the test.
- Embedded software is usually developed in parallel with hardware. Sometimes there are only few new hardware available for software testing.
- Embedded software is closely integrated with hardware. A fault may come from hardware rather than software.

Such challenges have led to wide adoption of **simulation-based testing** in embedded software industry. Some simulators can simulate all or part of embedded hardware, so engineers don't always perform tests on target boards. Depend on which part is simulated, this approach can be called X-in-the-loop, e.g., hardware-in-the-loop (HiL), software-in-the-loop (Sil), model-in-the-loop (MiL), processor-in-the-loop (PiL), etc. Examples of such simulators include Qemu and Tina. Some embedded software IDEs also contain simulators.

However, configure such simulators may take a lot of effort, especially when we configure different peripherals. On the other hand, any simulator cannot 100% mimic real hardware. Many tests of embedded software are still performed on real hardware (Garousi, Felderer, Karapıçak, & Yılmaz, 2018), as shown in Figure 56.

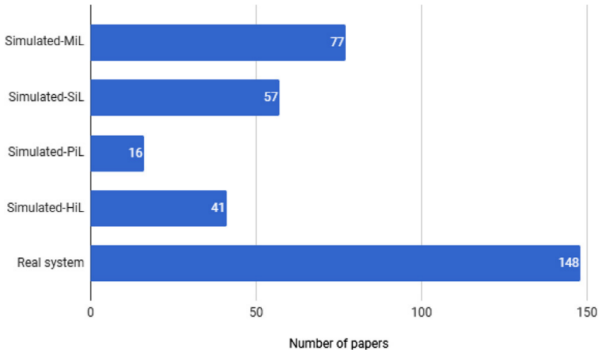


Figure 56: Papers in terms of using simulated or real SUTs (Garousi et al., 2018)

Although testing embedded software is difficult, it's still important in the research field of software testing, as shown by Figure 57. Many papers are related to the aerospace engineering, as shown in Figure 58.

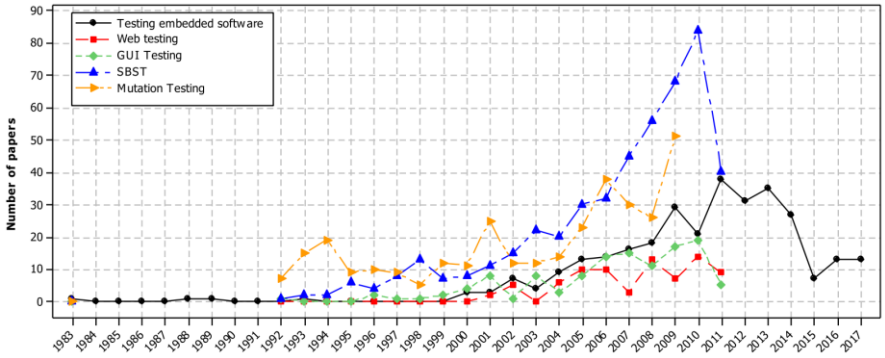


Figure 57: Growth of the Field (Testing Embedded Software) (Garousi et al., 2018)

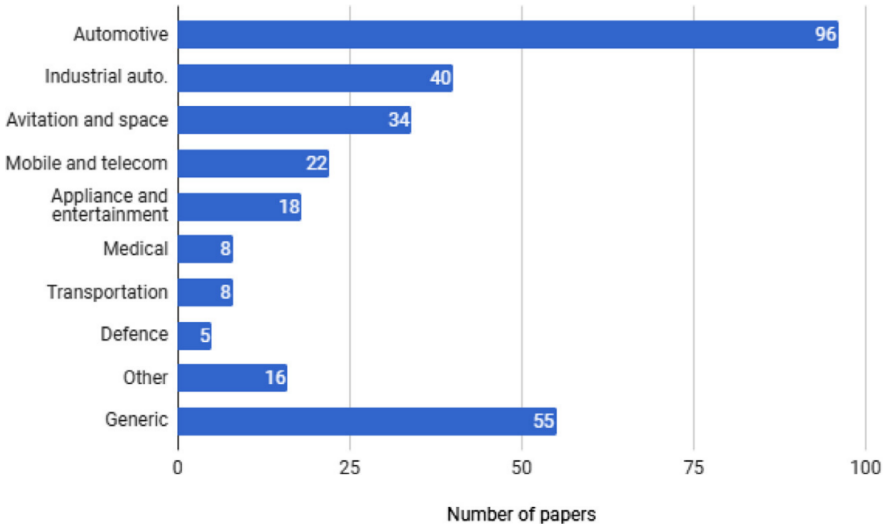


Figure 58: Applications of Embedded Software Testing (Garousi et al., 2018)

(Garousi et al., 2018) is a comprehensive survey of embedded software testing, though it has many errors in the references.

5.2. Traditional Test Case Generation Techniques

We briefly review existing automated test case generation techniques for correctness tests in this section. Two related surveys are (Utting, Pretschner, & Legeard, 2012) and (Anand et al., 2013). The goal of correctness tests can be

- Coverage, e.g., code coverage, state coverage, requirement coverage, etc.
- Test case specifications set by human, e.g., reach specific states.
- Spread test cases evenly in the input domain to achieve high coverage.
- Detect more faults.

During execution of test cases, faults can be detected by **test oracles**. A test oracle defines expected output of the software under certain preconditions. We can detect faults by comparing real output and expected output. In model-based testing, we can directly get expected outputs from a model so don't need test oracles anymore.

There are some standard languages for test cases of embedded software, including the TTCN-3 (Willcock, Deiß, Tobies, Keil, Engler & Schulz, 2011) and ATLAS2000 (Simpson, 2000).

5.2.1 Symbolic Execution

Symbolic execution analyses source code to generate test data that can achieve high code coverage (white box testing). In the process of code analysis, it uses symbolic variables to simulate execution of the software. At any point during symbolic execution, it maintains current symbolic variables, a **path constraint** on the symbolic variables, and a program counter. Only when inputs of the software can satisfy the path constraint, this path is feasible. In this way, symbolic execution can find all feasible paths, their path constraints and test inputs. Figure 59 shows an example of symbolic execution (Anand et al., 2013).

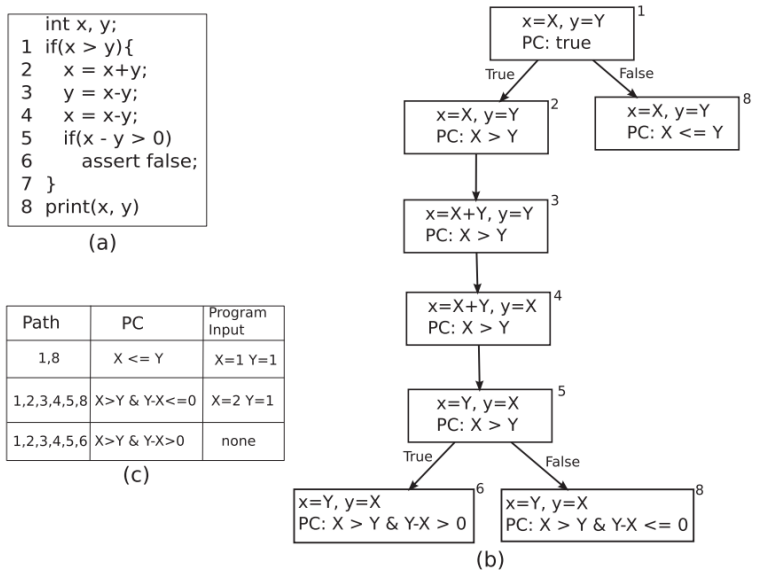


Fig. 1. (a) Code that swaps two integers, (b) the corresponding symbolic execution tree, and (c) test data and path constraints corresponding to different program paths.

Figure 59: An Example of Symbolic Execution

Although symbolic execution was proposed by King (1975), it only became feasible in the 21st century because of more powerful constraint solvers and computers. It still has some fundamental problems:

- **Path explosion**: Most real-world software has an extremely large number of paths, and many of these paths are infeasible. It takes too much time to symbolically execute all the paths.
- **Path divergence**: Most real-world software uses multiple programming languages, and parts of them may be available only in binary form. Users need to provide models for the problematic parts.
- **Complex constraints**: some path constraints include non-linear operations like multiplication, division, and mathematical functions like sin and log, which cannot be solved by available constraint solvers.

5.2.2 Model-Based Testing

Model-based testing is another white-box testing technique. Models can be built by engineers, or from reverse engineering of software under test. After that, model-based testing tools will derive test cases from the models to maximize some kinds of structural model coverage. The general process of model-based testing is shown in Figure 60.

We can also use mutated models to generate wrong test cases. These test cases can verify the fault handling mechanisms in programs.

There are many types of models. For embedded software, the most popular models are the finite state machine (FSM) and its variants (e.g., timed-FSM, stochastic-FSM). Apart from FSMs, other models like UML sequence diagrams, Method Definition Language (MeDeLa), logic formulars, labelled transition systems are also used.

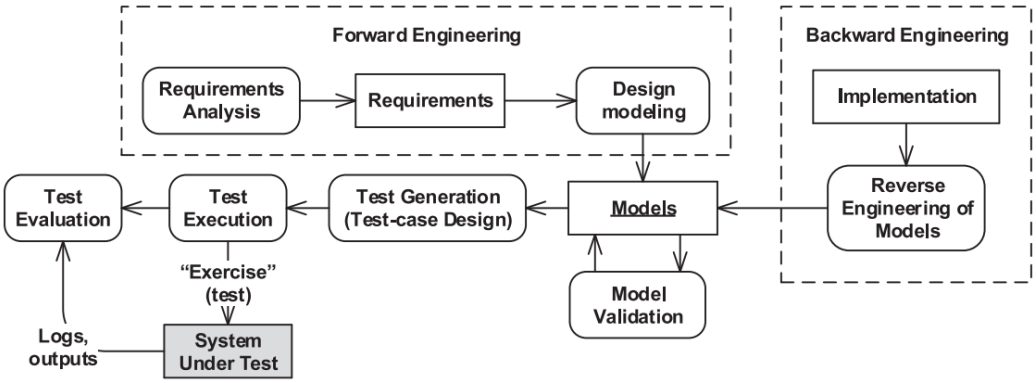


Figure 60: Process of Model-Based Testing (Garousi et al., 2018)

An example of model-based testing tools is the LDRA TBrun (LDRA, 2013). It can automatically analyses software under test and get a control flow graph. After that, it automatically generates testing functions from the control flow graph. The testing functions are compiled with the software under test, and they will send test inputs to the software during execution. However, TBrun is designed for unit testing, i.e., analyse some functions and test them. It cannot generate commands for multiple subsystems in a system test.

5.2.3 Random Testing

Empirical studies show that failure-causing inputs tend to form contiguous failure regions. Consequently, non-failure causing inputs also form contiguous non-failure regions. Therefore, test cases should be evenly spread across the input domain, rather than located around a specific region (Anand et al., 2013). Random testing (RT) takes this idea and generates test inputs randomly. Adaptive random testing is an enhancement to RT.

A famous random GUI testing tool is the Monkey provided by Android SDK (Patel, Srinivasan, Rahaman & Neamtiu, 2018). It can generate and execute test cases in a very high speed so it may achieve higher code coverage than some complex tools.

5.2.4 Search-Based Testing

In search-based testing, we first define a measurable fitness function that represents the test objectives. After that, we search a test case that maximize the fitness function. Commonly used search algorithms (Utting, Pretschner, & Legeard, 2012) include metaheuristic search, simulated annealing, and evolutionary algorithms (e.g., the genetic algorithm).

An example of search-based testing tools is the Sapienz (Mao, Harman & Jia, 2016). It was first developed as research in UCL, but massively deployed in Facebook after 17 months (Mao, 2018). Sapienz uses a multi-objective genetic algorithm to generate test cases with maximal fitness functions. If the source code of the Android app is available, Sapienz measures statement coverage as the fitness function during execution of the app. When the source code is unavailable, Sapienz measures method coverage or activity coverage instead. The workflow of Sapienz is shown in Figure 61.

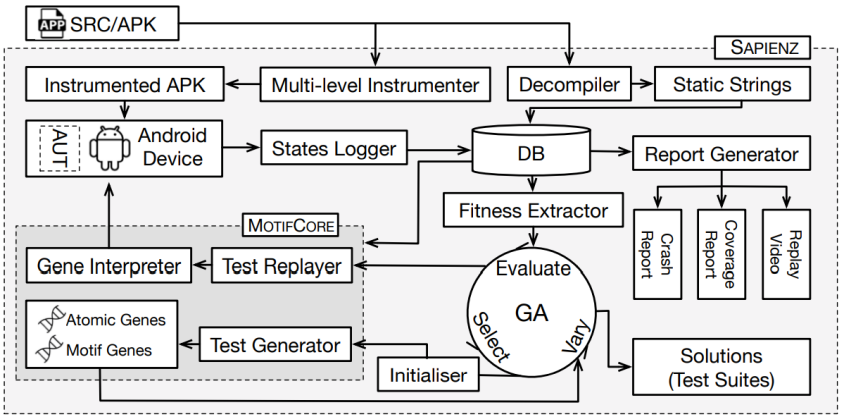


Figure 61: Sapienz Workflow

5.2.5 Other Traditional Techniques

Combinatorial interaction testing selects samples of input parameters that cover all combinations of the elements to be tested. It's like automated all-pair testing. Such samples can be generated by heuristic search or meta-heuristic search (Anand et al., 2013).

Model checking can verify whether a state machine satisfy a property represented by a temporal logic formula. If the FSM doesn't satisfy the property, model checkers can generate counter examples, which can be used as test cases. **Deductive theorem provers** can be used in a similar way (Utting, Pretschner, & Legeard, 2012). However, for both model checking and deductive theorem provers, a state transition model is needed.

5.3. Apply Planning in Correctness Testing

The basic idea is to let human specify testing goals, and an AI planner will generate a set of plans to satisfy the goals. To some extents, planning-based test case generation is a type of model-based testing: models of tested systems are implicitly represented in planning domains. If current state of software is different from the predicted state in the plan, a "fault" is detected.

However, this approach can only deal with "focused" goals such as reaching a state. It cannot achieve "ambiguous" objectives like reaching a high code coverage.

5.3.1 Historical Remarks

The first attempt to apply planning in automated testing is (Howe, Von Mayrhauser & Mraz, 1997). They used the UCPOP partial planner (Penberthy & Weld, 1992) to generate testing commands for the StorageTek robot tape library. More specifically,

- Step 1: The users indicate how many of different types of operations should be included in the plan. Based on the domain definition and users' specification, the **pre-processor** creates a problem definition (including an initial state and goal state) that would require using the indicated commands. For example, if the user requests 3 move operations to be accomplished, the pre-processor defines an initial state with at least 3 tapes in randomly selected positions and a goal state which specifies three new randomly selected locations for the tapes (Howe, Von Mayrhauser & Mraz, 1997).

- Step 2: Based on the planning domain definition and problem definition, the UCPOP **planner** generates a plan.
- Step 3: The **post-processor** transfers the plan to a sequence of commands.
- Note: To generate invalid test cases, the users can modify preconditions in the domain definition, or change rules in the pre-processor. This approach is like using mutated models in model-based testing in section 5.2.2.

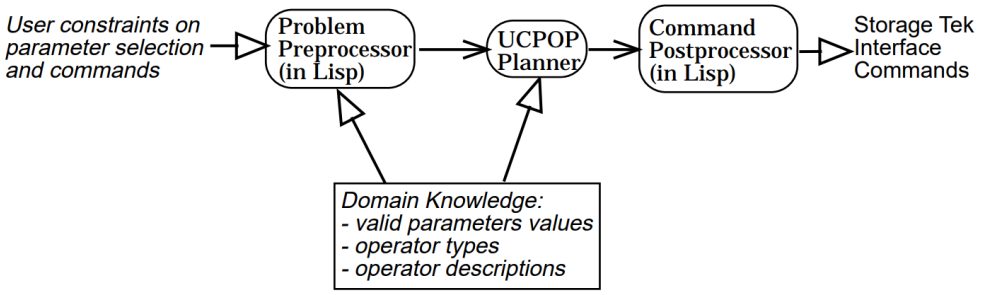


Figure 62: Overall Design of (Howe, Von Mayrhauser & Mraz, 1997)

It was a successful AI planning framework to generate both valid and invalid test cases. The pre-processor, post-processor, and planning domain definition with 18 operators were written in only 414 lines of code. However, 2 problems still exist: the UCPOP is not an efficient planner, and the planner only generates a single plan which doesn't cover too many states.

Memon, Pollack, and Soffa (2001) alleviated these problems. They found that GUI testing had the following properties:

- The GUI actions are inherently hierarchical in nature. For example, you need to open the "start" menu to click a button in the menu. As we discussed in chapter 2, hierarchical planning is more efficient.
- Hierarchical planning and partial planning can generate a set of plans, rather than a single plan, for an initial state and a goal. For example, you can choose different methods to decompose a task in the HTN planning, or different ways to complement a partial plan into a total plan. A set of plans (test cases) can cover more states.
- The GUI actions can be automatically extracted from GUI events. Therefore, users only need to specify preconditions and effects of these actions, rather than write the complete domain definition. It reduces the workload of modelling.

Utilizing the properties of GUI testing, Memon et al. implemented a test case generator called PATHS and used it to test the Microsoft WordPad. PATHS automatically derives operators from GUI events. After that, the users complement the domain definition and define a task as a problem definition. Given the task and domain definition, a hierarchical partial planner (like the AHA* planning in section 2.4.3) will generate a set of test cases. The workflow is shown in Figure 63.

Phase	Step	Test Designer	PATHS
Setup	1		Derive Hierarchical GUI Operators
	2	Define Preconditions and Effects of Operators	
Plan Generation	3	Identify a Task $\mathcal{T}$	
	4		Generate Test Cases for $\mathcal{T}$

Figure 63: Roles of Users and PATHS during Test Case Generation

Like PATHS, (Leitner and Bloem, 2006) also automatically derives planning problem and domain definitions from source code. Their system tests software written in a programming language called

Eiffel, which native support of preconditions and effects. If the derived effects of actions are not precise enough, the system can modified the effects according to execution results ($new\ eff \leftarrow old\ eff \cap execution\ result$). It's like the symbolic learning techniques in section 2.8.3. The planning and learning process is shown in Figure 64.

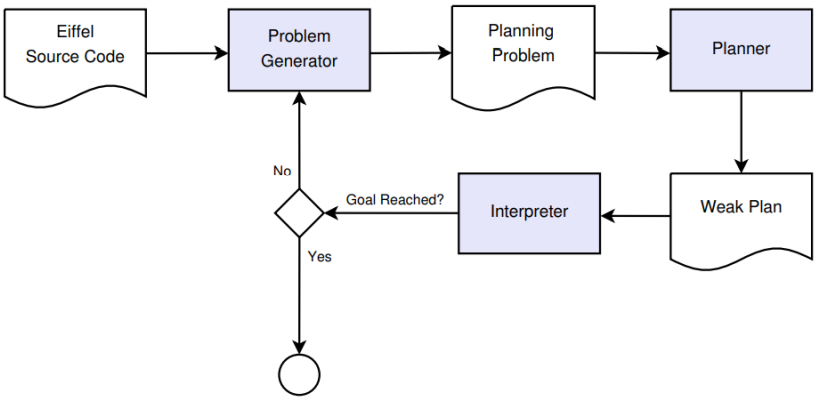


Figure 64: Planning and Learning Process in (Leitner and Bloem, 2006)

Later works to apply planning in correctness testing followed the methods mentioned above. An example is (Bozic, Tazl, & Wotawa, 2019), which uses AI planning to test chatbots.

It's important to mention that AI planning is more widely used in safety testing. Such tests have more "focused" goals like penetrating fireworks. Related works include (Sarraute, Buffet, & Hoffmann, 2012) and (Durkota & Lisý, 2014).

5.3.2 Discussion

As mentioned in the beginning of this section, AI planning is good at "focused" testing goals like reaching a state. It's not good at state/code coverage in nature. Although PATHS (Memon et al., 2001) tried to cover more states by generating a set of plans, it could not achieve as high coverage as other model-based testing techniques. For example, a depth-first or breadth-first directed graph traversal algorithm will cover more states than any planning algorithm within a given time.

In some scenarios, we need "focused" testing goals rather than higher state coverage. For example, if the state space is too large to cover, it's better to use testing goals derived from the requirements. In such scenarios, AI planning can find paths to the goals more quickly than traditional techniques in section 5.2.

For these "focused" tests, hierarchical planners and partial planners have a special advantage. As shown by PATHS, such planners can generate a set of different test cases for a goal, instead of a single test case. More test cases can cover more paths in a test.

Examples in section 5.3.1 mainly use classical planning. Can we use temporal planning and probabilistic planning to test complex software with durative actions, concurrency, and uncertainty? As mentioned in section 2.8.1, current first-order logic planning representations have poor prediction capability. It's not possible to fully represent behaviours of a complex system in several hundred lines of PDDL code. Therefore, if we use temporal/probabilistic planners to test complex systems, it's necessary to use replanning/plan-repair as a supplement. In this case, we need to use testing oracles, rather than prediction of planners, to identify faults.

A bottleneck of planning-based testing is to acquire the planning models, i.e., problem and domain definitions. As the tested systems become more complex, it takes more effort to write such

definitions by hand. PATHS automatically derived part of the definitions from source code. (Leitner and Bloem, 2006) also learnt the definitions from execution results. Learning seems to be an important direction in this field.

5.4. Apply Reinforcement Learning in Correctness Testing

This section reviews how to use reinforcement learning in test case generation. Most research use RL to maximize state coverage of the tested program. Such problem is like traversal for a directed graph, as shown in Figure 65. A node on the graph is a state of the tested software, and an edge is a state transition caused by an action. If all nodes and edges are known and we can start searching from any node, the traversal can be easily done by a depth-first or breath-first search algorithm.

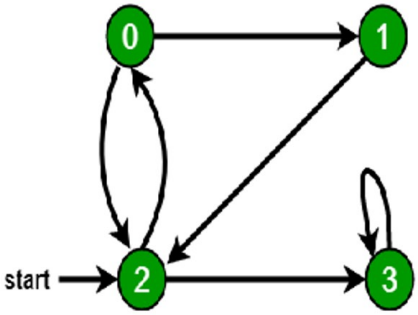


Figure 65: A Directed Graph

Again, life is not so easy. Real software testing has the following challenges:

- Behaviours of some software cannot be explained by a directed graph, a finite state machine or similar models.
- An action may have probabilistic outcomes.
- We won't know all nodes/edges if we don't have a model of the software.
- We can only start searching from initial state. Resetting to the initial state may take a lot of time.

To cover software states under these challenges, 2 approaches were proposed. The first approach uses a model-free RL agent to explore the state space, while the second approach learns a model of the tested software. Subsections 5.4.1 and 5.4.2 introduce the 2 approaches.

Subsection 5.4.3 briefly reviews other research to apply RL in software testing. (Durelli et al., 2019) and (Omri and Sinz, 2021) are 2 recent surveys about machine learning in software testing, which are related to this section.

5.4.1 "AI Spidering" – Model-Free RL to Explore Software

Traditional RL agents struggle to achieve a balance between exploration and exploitation. However, an "AI Spider" only focuses on exploration. As we discussed in section 3.5.2, exploration can be encouraged by giving the agent (count-based or curiosity-based) intrinsic rewards.

Let's start from count-based rewards. In the simplest version, the RL agent uses the Q-Learning algorithm in section 3.3.1 and receives an intrinsic reward after reaching any state s:

$$R = \frac{1}{N_s + c} \tag{5-1}$$

where N_s is the number of times that state s has been visited, and c is a constant. Compared with random testing in section 5.2.3, the Q-Learning agent records and chooses the actions that lead to more new states being discovered. Many researchers and developers adopt similar methods to explore the tested software:

- (Veanes, Roy & Campbell, 2006) records number of visits in edges of a state transition graph, which is constructed during the search. Humans write an implicit model (like a planning model) of the tested toy program so it knows what action can be executed under a specific state.
- (Bauersfeld & Vos, 2012) uses this method to test Microsoft Word's GUI.
- (Groce et al., 2012) uses the SARSA algorithm to test some toy programs.
- (Adamo, Khan, Koppula, & Bryce, 2018) uses this method to test GUIs of some Android Apps. Executable actions under a state are extracted by the Appium and UIAutomator tools. Under a small probability, the agent will restart search from the initial state to avoid being trapped in loops. Execution traces are recorded as test cases. For each test case, they calculate a hash value to avoid recording the similar case twice.
- (Vuong & Takada, 2018) this method to test GUIs of some Android Apps.
- Some commercial testing tools like Test.AI (test.ai, 2021) have similar features of the Q-Learning exploration. However, the technical details of these products are not clear.

The intrinsic rewards can also be given by a curiosity module. The curiosity module can be used as a state comparison function to differ different states. For example, there may be different news in the same home page of a GUI, as shown by Figure 66. Traditional methods may think that the home page contains many states, but a curiosity module will recognize the page as a single state.

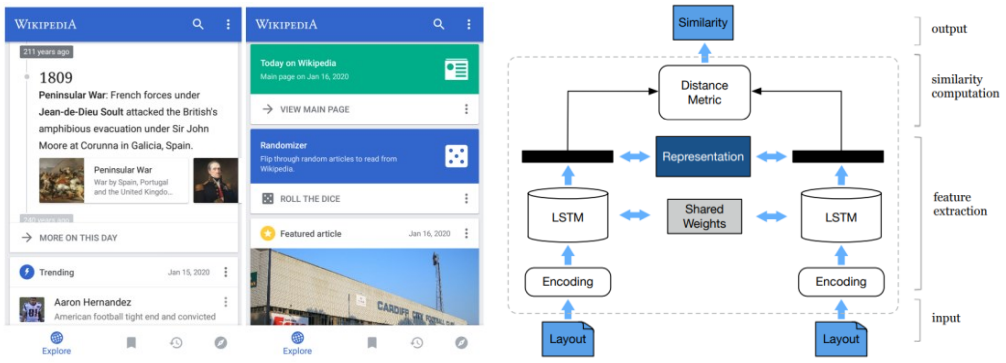


Figure 66: A Curiosity Module in an "AI Spider"

(Pan, Huang, Wang, Zhang & Li, 2020) uses the curiosity module in Figure 66 to compute rewards. Bytedance, the company who developed the popular application TikTok, is using an "AI spider" with a curiosity module in GUI testing. Their "Fastbot" tool (Bytedance, 2021) detects more than 50,000 crashes every month.

These "AI spiders" have been applied in industry and usually perform better than random testing. However, they cannot utilize experience of previous tests. For example, reward (5-1) is changing during a test, which means the environment is changing when the agent interacts with it. The optimal policy in the beginning of the test (most states not visited yet, dense reward) is different from the optimal policy in the end (most states visited, sparse reward). Therefore, a trained policy (optimal in the end) may not adapt to the beginning of the test very well.

At the same time, the "AI spiders" are mainly built for GUI or web testing, whose state/action space is discrete. The state space is also not very large (~100 pages for an Android app) so a tabular Q-Learning algorithm is enough. The agent can easily know what action can be executed from the current page. However, for more complex software, the approach may have problems.

5.4.2 Learn Models of the Software

Testing tools in this section interact with tested software to learn of model. As far as we know, most of them learn a Finite State Machine (FSM) model. (Fraser & Walkinshaw, 2015) also learns different classifiers, including neural networks and naive Bayes learners, from the tested software.

Learning a finite state machine is an old topic in computer science (Mohri, Rostamizadeh & Talwalkar, 2012). A state machine (or automata) can be learnt actively or passively. An active learning algorithm called L* (Angluin, 1987) has been widely used in domains from network protocol inference to functional confirmation testing of circuits. However, L* requires frequent reset to the initial state.

(Choi, Necula, & Sen, 2013) learns an FSM model from Android application and uses the model to guide test into unexplored parts of state space. They try to minimize number of resets during the learning process, which is shown in Figure 67. (Zheng et al, 2021) takes a similar approach. Stoat (Su et al., 2017), another popular Android testing tool, learns an FSM and then mutate it to generate test cases.

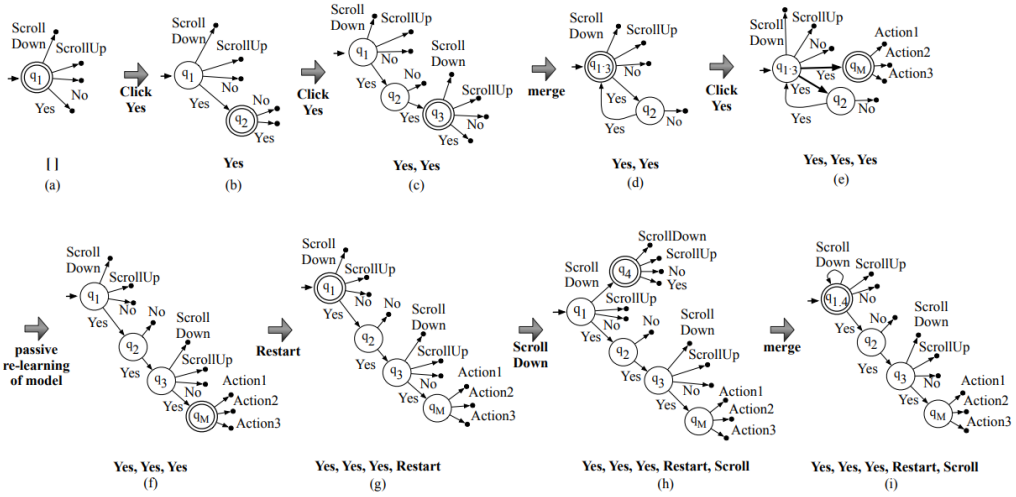


Figure 3: Progress of learning guided testing on *Sanity* example. A circle with solid line is used to denote a state in the model. The initial state is marked with a short incoming arrow. A circle with double line denotes the current model-state.

Figure 67: Learning Automata from Android Apps (Choi et al., 2013)

This approach can also be applied to embedded systems. For example, (Groz, Simao, Bremond & Oriat, 2018) infers an FSM from a heating management system with 3 C++ microcontrollers. Their learning algorithm can scale up to more than 1000 states.

Different from the “AI spiders”, models learnt from tested software can be reused in the next round of test. However, current learnt models in software testing are mainly finite state machines, which cannot be scaled up to very large state space and complex systems.

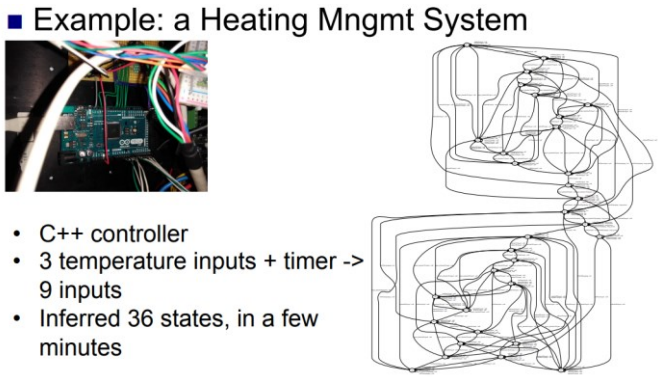


Figure 68: Learning Automata from Embedded Systems

5.4.3 Other Works to Apply RL in Software Testing

Reinforcement learning becomes more popular in game testing. It's reasonable since many RL research are performed using computer games. Because of limited collaboration capability of these agents (see section 3.5.5), they cannot be used as teammates of human players at this stage. Therefore, the only practical application of these agent will be testing the games. In this case, they need to achieve a balance between game scores and state coverage.

Wuji (Zheng et al., 2019) uses the multi-objective genetic algorithm with the standard A3C RL algorithm (see section 3.4.3) in web game testing. The genetic algorithm is used to modify weights of neural networks to achieve higher state coverage. The famous game company *Electronic Arts* also uses the PPO RL algorithm (see section 3.4.2) to test their games (Bergdahl, Gordillo, Tollmar, & Gisslén, 2020).

Apart from state coverage, RL agents can maximize other metrics in software testing. (Ahmad, Ashraf, Truscan, & Porres, 2019) uses the duelling DQN algorithm (see section 3.3.3) to do performance test so the reward is system workload. (Reichstaller, Eberhardinger, Knapp, Reif, & Gehlen, 2016) uses RL in risk-based testing. However, as far as we know, there is no research that directly uses code coverage as reward.

Apart from test case generation, RL can also be used to prioritize test cases in regression testing. (Spieker, Gotlieb, Marijan, & Mossige, 2017) is a successful work in this field. It calculates rewards by execution time and number of failures of test cases.

5.4.4 Discussion

Reinforcement learning can be thought as a search-based test case generation method:

- Unlike planning, reinforcement learning allows ambiguous goals like maximization of coverage. Such goals are important in correctness testing.
- Compared with model-based testing, RL needs less prior knowledge of the tested system.
- Compared with random testing, RL is guided to the unexplored regions.
- Compared with other search-based methods like the evolutionary algorithms, RL can generate valid action sequences with strong causal relationships.

To test software with small and discrete state/action space, maximizing state coverage is a reasonable goal. As shown in section 5.4.1, the tabular Q-learning algorithm has been successfully used to explore such state space. However, the learnt policy may not be very useful in the next round of test. To reuse experience of previous tests, it's better to learn an FSM model with up to

1000 states from the tested software, as shown in section 5.4.2. We can also include a coverage vector as part of the current state, so the rewards are not changing, and the learnt policy can be reused. As a compromise, it will significantly increase the size of the state space.

But how to explore a software with large or continuous state/action space? Covering such state/action space is very time-consuming or even impossible. One possible approach is to merge similar states and actions, which is identical to the equivalence partitioning in section 5.1.1. An example has been shown in section 5.4.1 where a curiosity module is used to differ from different states. Another approach is to use other metrics like code coverage, which may be more difficult to collect in practice. For example, in (Runeson, Heed, & Westrup, 2011), the testers insert line counting instructions in Java bytecode to measure real-time code coverage change on embedded devices. Every time a counting instruction is executed, the host PC will receive a message from the target board.

Table 1. Bytecode injection example. Injected rows are marked with an expression mark (!) at the start.

Original bytecode	Bytecode after injection
`public final example()V`	`public final example()V`
`  FRAME FULL [] []`	`! L0 (0)`
`  ALOAD 0`	`! LINENUMBER 1 L0`
`  GETFIELD h.f : Z`	`  FRAME FULL [] []`
`  IFNE L0`	`  ALOAD 0`
`  RETURN`	`  GETFIELD h.f : Z`
`L0 (6)`	`  IFNE L2`
`  RETURN MAXSTACK = 3`	`! L1 (5)`
`  MAXLOCALS = 3`	`! LINENUMBER 2 L1`
	`  RETURN`
	`  L2 (6)`
	`! LINENUMBER 3 L2`
	`  RETURN MAXSTACK = 3`
	`  MAXLOCALS = 3`

Figure 69: Inject Lind Counting Instructions in Java Bytecode

If deep reinforcement learning algorithms are used, their low sampling efficiency can be a significant problem. We cannot parallel collect a large amount of samples from a simulated environment. In practice, we may only have few target boards and the communication between the host and the targets takes a lot of time. The sampling efficiency may be improved if we learn a model of the tested embedded software.

5.5. Research Questions

5.5.1 Research Objective

The research objective is **suggesting an approach to automatically generate strong causal-related testing commands with limited prior knowledge for onboard software by designing and validation of a primitive planning/RL-based testing tool for the Delfi-PQ satellite.**

This research objective fills a gap of space system testing. As far as we know, current testing techniques in the space industry cannot automatically produce strong causal-related testing commands with limited prior knowledge:

- To automatically generate such commands, engineers usually write an FSM-like model of the system and use the program to traverse the model. It takes a lot of effort to define the prior knowledge.

- The model can also be extracted from the source code (like LDRA TBrun in section 5.2.2). However, static code analysis like symbolic execution is not good at complex software and mainly used in unit testing. It generates test inputs for specific functions.
- Random testing and evolutionary algorithms are not good at generating valid command sequences with strong causal relations.

On the other hand, planning and reinforcement learning are suitable to generate such commands, which is a sequential decision-making problem. They have been successfully used in GUI testing and web testing (section 5.3/5.4) but not used in onboard software yet. Some challenges remain, especially for testing onboard software:

- Planning can achieve “focused” goals like reaching a specific state. As shown in section 5.3.2, testers need to write an implicit planning model for the tested software, which may take some efforts. The predicting capability of the first-logic planning model is limited for complex onboard software.
- Reinforcement learning is good at “ambiguous” goals like state/code coverage. As shown in section 5.4.4, three fundamental problems of this approach are reusing experience, limited sampling from embedded software, and exploring software with large state/action space.

We can compare the primitive testing tool with a traditional model-based testing tool (commercial or developed by ourselves) to validate this approach. If we achieve the research objective, we may significantly reduce human workload in onboard software testing and increase the test coverage.

5.5.2 Research Questions about the Testing Goal

RQ1 What’s the goal of testing command generation?

Three possible goals are identified: maximize code coverage, maximize state coverage, or reach some “focused” goals set by human.

RQ1.1 If code coverage is the goal, how much effort is needed to collect code coverage change caused by a single command?

For tested software with large action/state space, code coverage is a promising goal. To understand how a specific command affects code coverage, we need to measure code coverage change caused by the command. However, as shown in section 5.4.4, such measurement is not easy. Therefore, we need to know:

RQ1.1.1 What software/hardware tool can get such coverage change from the microcontrollers of PQ?

Such as LDRA cover, Segger J-Trace, VectorCAST…

RQ1.1.2 How much money do we need to pay for these tools?

RQ1.1.3 Do we need to modify the tools, or even develop something by ourselves to measure such change?

RQ1.1.4 Does the same method work for other small satellites?

RQ1.2 If state coverage is the goal, how can we differ one state from another state for the Delfi-PQ onboard software?

It's easy to differ one state from another in GUI or web testing: one page layout is identical to a discrete state. However, it's difficult to identify a discrete state from telemetry of some onboard software. For example, most of parameters in the telemetry of ADB are continuous. Several possible solutions exist:

RQ1.2.1 Can we use a special type of state coverage that is easier to measure?

An example of such metrics is to cover different types of commands and replies?

RQ1.2.2 Can we use a curiosity module (or similar things) to compute the "similarity" of continuous telemetry parameters?

This means that we try to distinguish states with continuous telemetry parameters.

RQ1.3 If we set "focused" goals, how can we generate representative goals that can sufficiently test the software?

RQ1.3.1 Can we do that by imitating behaviours of human operators (imitation learning)?

It's also common to imitate human behaviours in software testing.

RQ1.3.2 Can users set constraints of the goals, and then a goal generator sets the goals automatically?

Like the approach in section 5.3.1.

5.5.3 Research Questions about Prior Knowledge

RQ2 How much prior knowledge needs to be encoded?

In general, we prefer a method that need less prior knowledge, and the prior knowledge is easy to encode. It will reduce workload of testers.

RQ2.1 The Delfi-PQ mission defines formats of commands and replies in an XML file according to the XTCE standard (CCSDS 660.0). Can a testing tool read the format automatically?

The XML file records the formats of all commands and replies for each subsystem. However, it doesn't contain predictions and effects of each command.

RQ2.2 Can the testing tool learn preconditions and effects of a command during execution?

Some commands can only be successfully executed under some preconditions. Effects of a command include type of reply, state change, and code coverage change. By acquiring these preconditions and effects, the tool learns a model of the onboard software. It's helpful but not necessary.

RQ2.2.1 What kind of model is suitable for learning preconditions, type of reply, and state change?

The answer may be a finite state machine, a planning model, or a neural network.

RQ2.2.2 What kind of model is suitable for learning code coverage change?

The answer may be a neural network, whose inputs include a code coverage vector and a command, and output is a new code coverage vector. The size of code coverage vector depends on resolution set by human (doesn't change with code).

5.5.4 Research Questions about Testing Algorithm

RQ3 What algorithm is suitable for testing command generation?

The answer strongly depends on the goal and the prior knowledge we have.

RQ3.1 For an RL algorithm, how can we reuse previous experience?

RQ3.2 For an RL algorithm, how can we improve sampling efficiency on embedded devices?

RQ3.3 For a planning algorithm, how can we improve the prediction capability of the first-order logic model?

5.5.5 Research Questions about Testing Environment

RQ4 Do we test the software on target hardware, in a simulator, or in a hardware-in-the-loop environment?

Probably on target hardware because we don't have a simulator yet. Texas Instrument doesn't provide a simulator for the MSP432 microcontroller and prefers that customers evaluate their solution with the physical hardware itself.

RQ5 How do we set the testing environment when the EGSE is the bus master?

We can send commands and receive replies by the EGSE. However, we may consider other connections.

RQ5.1 Shall we connect the JTAG pins to collect debug information, including code coverage?

Unfortunately, there is only one Segger J-LINK PRO connector.

RQ5.2 For the COMMS subsystem, shall we use the wireless communication?

RQ5.3 Shall we mimic other inputs, such as magnetic field and sunlight?

RQ6 How do we set the testing environment when the OBC is the bus master?

We can send commands and receive replies by the COMMS. However, we may consider other connections.

RQ6.1 Can we use the EGSE to hear messages over the bus, or even insert wrong messages?

RQ6.2 Can we use the EPS to manage electricity of the satellite?

RQ7 How can we ensure the generated testing inputs will not damage the satellite?

5.5.6 Research Questions about Fault Detection

RQ8 How shall we design the test oracles?

Bibliography

Adamo, D., Khan, M. K., Koppula, S., & Bryce, R. (2018, November). Reinforcement learning for android gui testing. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (pp. 2-8).

Ahmad, T., Ashraf, A., Truscan, D., & Porres, I. (2019, August). Exploratory performance testing using reinforcement learning. In 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) (pp. 156-163). IEEE.

Allen, J. F., & Koomen, J. A. (1983, August). Planning using a temporal world model. In Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2 (pp. 741-747).

Ammann, P., & Offutt, J. (2016). Introduction to software testing. Cambridge University Press.

Anand, S., Burke, E. K., Chen, T. Y., Clark, J., Cohen, M. B., Grieskamp, W., ... & Zhu, H. (2013). An orchestrated survey of methodologies for automated software test case generation. Journal of Systems and Software, 86(8), 1978-2001.

Ai-Chang, M., Bresina, J., Charest, L., Chase, A., Hsu, J. J., Jonsson, A., ... & Maldague, P. F. (2004). Mapgen: mixed-initiative planning and scheduling for the mars exploration rover mission. IEEE Intelligent Systems, 19(1), 8-12.

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., ... & Zaremba, W. (2017). Hindsight experience replay. arXiv preprint arXiv:1707.01495.

Angluin, D. (1987). Learning regular sets from queries and counterexamples. Information and computation, 75(2), 87-106.

Arentoft, M. M., Parrod, Y., Stader, J., Stokes, I., & Vadon, H. (1991). OPTIMUM-AIV: A planning and scheduling system for spacecraft AIV. Telematics and Informatics, 8(4), 239-252.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. Machine learning, 47(2), 235-256.

Bacchus, F., & Kabanza, F. (2000). Using temporal logics to express search control knowledge for planning. Artificial intelligence, 116(1-2), 123-191.

Bauersfeld, S., & Vos, T. (2012, September). A reinforcement learning approach to automated gui robustness testing. In Fast abstracts of the 4th symposium on search-based software engineering (SSBSE 2012) (pp. 7-12). IEEE.

Behzadan, V., & Munir, A. (2017, July). Vulnerability of deep reinforcement learning to policy induction attacks. In International Conference on Machine Learning and Data Mining in Pattern Recognition (pp. 262-275). Springer, Cham.

Bellemare, M. G., Dabney, W., & Munos, R. (2017, July). A distributional perspective on reinforcement learning. In International Conference on Machine Learning (pp. 449-458). PMLR.

Bengio, Y., Lecun, Y., & Hinton, G. (2021). Deep learning for AI. Communications of the ACM, 64(7), 58-65.

Benton, J., Coles, A., & Coles, A. (2012, May). Temporal planning with preferences and time-dependent continuous costs. In Twenty-Second International Conference on Automated Planning and Scheduling.

Bergdahl, J., Gordillo, C., Tollmar, K., & Gisslén, L. (2020, August). Augmenting automated game testing with deep reinforcement learning. In 2020 IEEE Conference on Games (CoG) (pp. 600-603). IEEE.

Bertoli, P., Cimatti, A., Dal Lago, U., & Pistore, M. (2003, March). Extending PDDL to nondeterminism, limited sensing and iterative conditional plans. In ICAPS workshop on pddl, informal proceedings (pp. 15-24).

Blum, A. L., & Furst, M. L. (1997). Fast planning through planning graph analysis. Artificial intelligence, 90(1-2), 281-300.

Bonet, B., & Geffner, H. (1999, September). Planning as heuristic search: New results. In European Conference on Planning (pp. 360-372). Springer, Berlin, Heidelberg.

Bouwmeester, J., van der Linden, S. P., Povalac, A., & Gill, E. K. A. (2018). Towards an innovative electrical interface standard for PocketQubes and CubeSats. Advances in Space Research, 62(12), 3423-3437.

Bouwmeester, J., Radu, S., Uludag, M. S., Chronas, N., Speretta, S., Menicucci, A., & Gill, E. K. A. (2020). Utility and constraints of PocketQubes. CEAS Space Journal, 12(4), 573-586.

Bozic, J., Tazl, O. A., & Wotawa, F. (2019, April). Chatbot testing using AI planning. In 2019 IEEE International Conference On Artificial Intelligence Testing (AITest) (pp. 37-44). IEEE.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language models are few-shot learners. arXiv preprint arXiv:2005.14165.

Burda, Y., Edwards, H., Storkey, A., & Klimov, O. (2018). Exploration by random network distillation. arXiv preprint arXiv:1810.12894.

Bytedance. (2021). Fastbot_ios. https://github.com/bytedance/Fastbot_iOS

Chapelle, O., & Li, L. (2011). An empirical evaluation of thompson sampling. Advances in neural information processing systems, 24, 2249-2257.

Chien, S. (2012). A generalized timeline representation, services, and interface for automating space mission operations. In SpaceOps 2012 (p. 1275459).

Chien, S., Doubleday, J., Thompson, D. R., Wagstaff, K. L., Bellardo, J., Francis, C., ... & Piug-Suari, J. (2017). Onboard autonomy on the intelligent payload experiment cubesat mission. Journal of Aerospace Information Systems, 14(6), 307-315.

Choi, W., Necula, G., & Sen, K. (2013). Guided gui testing of android apps with minimal restart and approximate learning. Acm Sigplan Notices, 48(10), 623-640.

Clement, B. (2013). AI Planning for Space. http://media.aiai.ed.ac.uk/Project/AIPLAN/video/Feature-Brad-Clement-AI-Planning-for-Space.mp4

Cividanes, F., Ferreira, M., & Kucinskis, F. (2019). On-board Automated Mission Planning for Spacecraft Autonomy: A Survey. IEEE Latin America Transactions, 17(06), 884-896.

Cividanes, F. D. S., Ferreira, M. G. V., & de Novaes Kucinskis, F. (2021). An Extended HTN Language for Onboard Planning and Acting Applied to a Goal-Based Autonomous Satellite. IEEE Aerospace and Electronic Systems Magazine, 36(8), 32-50.

Currie, K., & Tate, A. (1991). O-plan: the open planning architecture. Artificial intelligence, 52(1), 49-86.

Dabney, W., Ostrovski, G., Silver, D., & Munos, R. (2018, July). Implicit quantile networks for distributional reinforcement learning. In International conference on machine learning (pp. 1096-1105). PMLR.

Dabney, W., Rowland, M., Bellemare, M. G., & Munos, R. (2018, April). Distributional reinforcement learning with quantile regression. In Thirty-Second AAAI Conference on Artificial Intelligence.

Dasgupta, I., Wang, J., Chiappa, S., Mitrovic, J., Ortega, P., Raposo, D., ... & Kurth-Nelson, Z. (2019). Causal reasoning from meta-reinforcement learning. arXiv preprint arXiv:1901.08162.

Deisenroth, M., & Rasmussen, C. E. (2011). PILCO: A model-based and data-efficient approach to policy search. In Proceedings of the 28th International Conference on machine learning (ICML-11) (pp. 465-472).

Do, M., & Kambhampati, S. (2003). Sapa: A multi-objective metric temporal planner. Journal of Artificial Intelligence Research, 20, 155-194.

Drabble, B., Dalton, J., & Tate, A. (1997). Repairing plans on-the-fly. In Proceedings of the NASA Workshop on Planning and Scheduling for Space.

Durelli, V. H., Durelli, R. S., Borges, S. S., Endo, A. T., Eler, M. M., Dias, D. R., & Guimarães, M. P. (2019). Machine learning applied to software testing: A systematic mapping study. IEEE Transactions on Reliability, 68(3), 1189-1212.

Durkota, K., & Lisý, V. (2014). Computing Optimal Policies for Attack Graphs with Action Failures and Costs. In STAIRS (pp. 101-110).

Emil Keyder, J¨org Hoffmann, and Patrik Haslum. Semi-relaxed plan heuristics. In Blai Bonet, Lee McCluskey, Jos´e Reinaldo Silva, and Brian Williams, editors, Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS 2012). AAAI Press, 2012.

Erol, K., Hendler, J. A., & Nau, D. S. (1994, June). UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning. In Aips (Vol. 94, pp. 249-254).

Espeholt, L., Soyer, H., Munos, R., Simonyan, K., Mnih, V., Ward, T., ... & Kavukcuoglu, K. (2018, July). Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures. In International Conference on Machine Learning (pp. 1407-1416). PMLR.

Eyerich, P., Mattmüller, R., & Röger, G. (2009, October). Using the context-enhanced additive heuristic for temporal and numeric planning. In Nineteenth International Conference on Automated Planning and Scheduling.

Finn, C., Abbeel, P., & Levine, S. (2017, July). Model-agnostic meta-learning for fast adaptation of deep networks. In International Conference on Machine Learning (pp. 1126-1135). PMLR.

Fikes, R. E., & Nilsson, N. J. (1971). STRIPS: A new approach to the application of theorem proving to problem solving. Artificial intelligence, 2(3-4), 189-208.

Fortunato, M., Azar, M. G., Piot, B., Menick, J., Osband, I., Graves, A., ... & Legg, S. (2017). Noisy networks for exploration. arXiv preprint arXiv:1706.10295.

Fox, M., & Long, D. (2003). PDDL2. 1: An extension to PDDL for expressing temporal planning domains. Journal of artificial intelligence research, 20, 61-124.

Fraser, G., & Walkinshaw, N. (2015). Assessing and generating test sets in terms of behavioural adequacy. Software Testing, Verification and Reliability, 25(8), 749-780.

Fratini, S., Cesta, A., De Benedictis, R., Orlandini, A., & Rasconi, R. (2011). Apsi-based deliberation in goal oriented autonomous controllers. ASTRA, 11.

Fuchs, J. J., Gasquet, A., Olalainty, B., & Currie, K. (1990, June). plan ERS-1: an expert planning system for generating spacecraft mission plans. In 1991 First International Conference on Expert Planning Systems (pp. 70-75). IET.

Fujimoto, S., Hoof, H., & Meger, D. (2018, July). Addressing function approximation error in actor-critic methods. In International Conference on Machine Learning (pp. 1587-1596). PMLR.

Fukushima, Y., & Mita, M. (2011, August). A new approach to autonomous onboard mission replanning using orthogonal array design. In 2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology (pp. 43-50). IEEE.

Garousi, V., Felderer, M., Karapıçak, Ç. M., & Yılmaz, U. (2018). Testing embedded software: A survey of the literature. Information and Software Technology, 104, 14-45.

Gerevini, A., & Serina, I. (2000, April). Fast Plan Adaptation through Planning Graphs: Local and Systematic Search Techniques. In AIPS (pp. 112-121).

Ghallab, M., & Laruelle, H. (1994, June). Representation and Control in IxTeT, a Temporal Planner. In Aips (Vol. 1994, pp. 61-67).

Ghallab, M., Nau, D., & Traverso, P. (2004). Automated Planning: theory and practice. Elsevier.

Ghallab, M., Nau, D., & Traverso, P. (2014). The actor's view of automated planning and acting: A position paper. Artificial Intelligence, 208, 1-17.

Ghallab, M., Nau, D., & Traverso, P. (2016). Automated planning and acting. Cambridge University Press.

Grinsztajn, N., Ferret, J., Pietquin, O., Preux, P., & Geist, M. (2021). There Is No Turning Back: A Self-Supervised Approach for Reversibility-Aware Reinforcement Learning. arXiv preprint arXiv:2106.04480.

Groce, A., Fern, A., Pinto, J., Bauer, T., Alipour, A., Erwig, M., & Lopez, C. (2012, November). Lightweight automated testing with adaptation-based programming. In 2012 IEEE 23rd International Symposium on Software Reliability Engineering (pp. 161-170). IEEE.

Groz, R., Simao, A., Bremond, N., & Oriat, C. (2018, May). Revisiting AI and testing methods to infer FSM models of black-box systems. In 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST) (pp. 16-19). IEEE.

Guzman, C., Castejon, P., Onaindia, E., & Frank, J. (2015). Reactive execution for solving plan failures in planning control applications. Integrated Computer-Aided Engineering, 22(4), 343-360.

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018, July). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In International conference on machine learning (pp. 1861-1870). PMLR.

Hammond, K. J. (1990). Explaining and repairing plans that fail. Artificial intelligence, 45(1-2), 173-228.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., ... & Silver, D. (2018, April). Rainbow: Combining improvements in deep reinforcement learning. In Thirty-second AAAI conference on artificial intelligence.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., ... & Gruslys, A. (2018, April). Deep q-learning from demonstrations. In Thirty-second AAAI conference on artificial intelligence.

Ho, J., & Ermon, S. (2016). Generative adversarial imitation learning. Advances in neural information processing systems, 29, 4565-4573.

Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., & Alford, R. (2020, April). HDDL: An extension to PDDL for expressing hierarchical planning problems. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 34, No. 06, pp. 9883-9891).

Hoffmann, J., and Nebel, B. 2001. The FF Planning System: Fast Plan Generation through Heuristic Search. Journal of Artificial Intelligence Research, 14: 253–302.

Hopgood, A. A. (1993). Knowledge-based systems for engineers and scientists. CRC Press, Inc.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., Van Hasselt, H., & Silver, D. (2018). Distributed prioritized experience replay. arXiv preprint arXiv:1803.00933.

Howe, A. E., Von Mayrhauser, A., & Mraz, R. T. (1997). Test case generation as an AI planning problem. In Knowledge-Based Software Engineering (pp. 77-106). Springer, Boston, MA.

Hunsberger, L., Posenato, R., & Combi, C. (2012). The dynamic controllability of conditional stns with uncertainty. arXiv preprint arXiv:1212.2005.

Hussein, A., Gaber, M. M., Elyan, E., & Jayne, C. (2017). Imitation learning: A survey of learning methods. ACM Computing Surveys (CSUR), 50(2), 1-35.

Ingrand, F., & Ghallab, M. (2017). Deliberation for autonomous robots: A survey. Artificial Intelligence, 247, 10-44.

Islam, R., Henderson, P., Gomrokchi, M., & Precup, D. (2017). Reproducibility of benchmarked deep reinforcement learning tasks for continuous control. arXiv preprint arXiv:1708.04133.

Jaderberg, M., Mnih, V., Czarnecki, W. M., Schaul, T., Leibo, J. Z., Silver, D., & Kavukcuoglu, K. (2016). Reinforcement learning with unsupervised auxiliary tasks. arXiv preprint arXiv:1611.05397.

Jiménez, S., De La Rosa, T., Fernández, S., Fernández, F., & Borrajo, D. (2012). A review of machine learning for automated planning. The Knowledge Engineering Review, 27(4), 433-467.

Johnston, M. D. (1990, January). Spike: Ai scheduling for nasa's hubble space telescope. In Sixth Conference on Artificial Intelligence for Applications (pp. 184-185). IEEE Computer Society.

Kautz, H., & Selman, B. (1999, June). Unifying SAT-based and graph-based planning. In IJCAI (Vol. 99, pp. 318-325).

King, J. C. (1975). A new approach to program testing. ACM Sigplan Notices, 10(6), 228-233.

Kirk, R., Zhang, A., Grefenstette, E., & Rocktäschel, T. (2021). A Survey of Generalisation in Deep Reinforcement Learning. arXiv preprint arXiv:2111.09794.

Kocsis, L., & Szepesvári, C. (2006, September). Bandit based monte-carlo planning. In European conference on machine learning (pp. 282-293). Springer, Berlin, Heidelberg.

Konda, V. R., & Tsitsiklis, J. N. (2000). Actor-critic algorithms. In Advances in neural information processing systems (pp. 1008-1014).

Kulkarni, T. D., Narasimhan, K., Saeedi, A., & Tenenbaum, J. (2016). Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. Advances in neural information processing systems, 29, 3675-3683.

Kvarnström, J., & Doherty, P. (2000). TALplanner: A temporal logic based forward chaining planner. Annals of mathematics and Artificial Intelligence, 30(1), 119-169.

Ldra. (2013). LDRA_TBrun. https://ldra.com/wp-content/uploads/2017/07/LDRA_TBrun-v5.2.pdf

Leitner, A., & Bloem, R. (2005). Automatic testing through planning. Technische Universität Graz, Institute for Software Technology, Tech. Rep.

Levine, S., & Koltun, V. (2013, May). Guided policy search. In International conference on machine learning (pp. 1-9). PMLR.

Lgvaz. (2017). Why Does Q-Learning Overestimate Action Values? Stackexchange. https://stats.stackexchange.com/questions/277442/why-does-q-learning-overestimate-action-values

Likhachev, M., Thrun, S., & Gordon, G. J. (2004). Planning for markov decision processes with sparse stochasticity. Advances in neural information processing systems, 17, 785-792.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... & Wierstra, D. (2015). Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971.

Luce, R. D. (2012). Individual choice behavior: A theoretical analysis. Courier Corporation.

Markov Decision Process. (2021). Wikipedia. https://en.wikipedia.org/wiki/Markov_decision_process

Malte Helmert, Patrik Haslum, and J¨org Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In Mark Boddy, Maria Fox, and Sylvie Thiebaux, editors, Proceedings of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07), pages 176–183, Providence, Rhode Island, USA, 2007. Morgan Kaufmann.

Mao, K., Harman, M., & Jia, Y. (2016, July). Sapienz: Multi-objective automated testing for android applications. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 94-105).

Mao Ke. (2018). Sapienz: Intelligent Automated Software Testing at Scale. https://engineering.fb.com/2018/05/02/developer-tools/sapienz-intelligent-automated-software-testing-at-scale/

Marthi, B., Russell, S. J., & Wolfe, J. A. (2008, December). Angelic Hierarchical Planning: Optimal and Online Algorithms. In ICAPS (pp. 222-231).

McDermott, D. (2003, June). The formal semantics of processes in PDDL. In Proc. ICAPS Workshop on PDDL (pp. 101-155).

Memon, A. M., Pollack, M. E., & Soffa, M. L. (2001). Hierarchical GUI test case generation using automated planning. IEEE transactions on software engineering, 27(2), 144-155.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. nature, 518(7540), 529-533.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., ... & Kavukcuoglu, K. (2016, June). Asynchronous methods for deep reinforcement learning. In International conference on machine learning (pp. 1928-1937). PMLR.

Mohalik, S. K., Jayaraman, M. B., Badrinath, R., & Feljan, A. V. (2018). HIPR: An Architecture for Iterative Plan Repair in Hierarchical Multi-agent Systems. J. Comput., 13(3), 351-359.

Mohri, M., Rostamizadeh, A., & Talwalkar, A. (2018). Foundations of machine learning. MIT press.

Moerland, T. M., Broekens, J., & Jonker, C. M. (2020). A framework for reinforcement learning and planning. arXiv preprint arXiv:2006.15009.

Moerland, T. M., Broekens, J., & Jonker, C. M. (2020). Model-based reinforcement learning: A survey. arXiv preprint arXiv:2006.16712.

Muscettola, N., Nayak, P. P., Pell, B., & Williams, B. C. (1998). Remote agent: To boldly go where no AI system has gone before. Artificial intelligence, 103(1-2), 5-47.

NAOC. (2018). Einstein Probe Mission. http://ep.nao.cas.cn/

Nau, D., Cao, Y., Lotem, A., & Munoz-Avila, H. (1999, July). SHOP: Simple hierarchical ordered planner. In Proceedings of the 16th international joint conference on Artificial intelligence-Volume 2 (pp. 968-973).

Nau, D. S., Au, T. C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D., & Yaman, F. (2003). SHOP2: An HTN planning system. Journal of artificial intelligence research, 20, 379-404.

Nau, D. S. (2007). Current trends in automated planning. AI magazine, 28(4), 43-43.

Nau, D. (2021, April 7). Automated Planning and Acting: Lecture Slides and Errata List. http://www.cs.umd.edu/users/nau/apa/slides/

Omri, S., & Sinz, C. (2021). Machine Learning Techniques for Software Quality Assurance: A Survey. arXiv preprint arXiv:2104.14056.

Osanlou, K., Frank, J., Benton, J., Bursuc, A., Guettier, C., Jacopin, E., & Cazenave, T. (2021). Time-based Dynamic Controllability of Disjunctive Temporal Networks with Uncertainty: A Tree Search Approach with Graph Neural Network Guidance. arXiv preprint arXiv:2108.01068.

Palmieri, M. (2013). System Testing in a Simulated Environment.

Pan, M., Huang, A., Wang, G., Zhang, T., & Li, X. (2020, July). Reinforcement learning based curiosity-driven testing of android applications. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 153-164).

Patel, P., Srinivasan, G., Rahaman, S., & Neamtiu, I. (2018, May). On the effectiveness of random testing for Android: or how i learned to stop worrying and love the monkey. In Proceedings of the 13th International Workshop on Automation of Software Test (pp. 34-37).

Pateria, S., Subagdja, B., Tan, A. H., & Quek, C. (2021). Hierarchical Reinforcement Learning: A Comprehensive Survey. ACM Computing Surveys (CSUR), 54(5), 1-35.

Pathak, D., Agrawal, P., Efros, A. A., & Darrell, T. (2017, July). Curiosity-driven exploration by self-supervised prediction. In International conference on machine learning (pp. 2778-2787). PMLR.

Patra, S., Mason, J., Kumar, A., Ghallab, M., Traverso, P., & Nau, D. (2020, June). Integrating acting, planning, and learning in hierarchical operational models. In Proceedings of the International Conference on Automated Planning and Scheduling (Vol. 30, pp. 478-487).

Patrick Haslum and Hector Geffner. Admissible heuristics for optimal planning. In S. Chien, R. Kambhampati, and C. Knoblock, editors, Proceedings of the 5th International Conference on Artificial Intelligence Planning Systems (AIPS-00), pages 140–149, Breckenridge, CO, 2000. AAAI Press, Menlo Park.

Pfau, D., & Vinyals, O. (2016). Connecting generative adversarial networks and actor-critic methods. arXiv preprint arXiv:1610.01945.

Penberthy, J. S., & Weld, D. S. (1992). UCPOP: A Sound, Complete, Partial Order Planner for ADL. Kr, 92, 103-114.

Penberthy, J. S., & Weld, D. S. (1994, July). Temporal planning with continuous change. In Aaai (Vol. 94, p. 1010).

Rabideau, G., Chien, S., Galer, M., Nespoli, F., & Costa, M. (2017). Managing Spacecraft Memory Buffers with Concurrent Data Collection and Downlink. Journal of Aerospace Information Systems, 14(12), 637-651.

Rabideau, G., Wong, V., Gaines, D., Agrawal, J., Chien, S., Kuhn, S., ... & Biehl, J. (2020). Onboard automated scheduling for the mars 2020 rover. In Proceedings of the International Symposium on Artificial Intelligence, Robotics and Automation for Space, i-SAIRAS.

Radu, S., Uludag, M. S., Speretta, S., Bouwmeester, J., Gill, E., & Foteinakis, N. C. (2018). Delfi-PQ: The first pocketqube of Delft University of Technology. In 69th International Astronautical Congress, Bremen, Germany, IAC.

Rajeswaran, A., Lowrey, K., Todorov, E., & Kakade, S. (2017). Towards generalization and simplicity in continuous control. arXiv preprint arXiv:1703.02660.

Reichstaller, A., Eberhardinger, B., Knapp, A., Reif, W., & Gehlen, M. (2016, October). Risk-based interoperability testing using reinforcement learning. In IFIP International Conference on Testing Software and Systems (pp. 52-69). Springer, Cham.

Rui Xu, Chao Chen, Pingyuan Cui, Shengying Zhu, & Fan Xu. (2019). Research on spacecraft autonomous mission plan repair. Journal of Astronautics, 040(007), 733-741.

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. nature, 323(6088), 533-536.

Rummery, G. A., & Niranjan, M. (1994). On-line Q-learning using connectionist systems (Vol. 37, p. 20). Cambridge, England: University of Cambridge, Department of Engineering.

Runeson, P., Heed, P., & Westrup, A. (2011, June). A Factorial Experimental Evaluation of Automated Test Input Generation. In International Conference on Product Focused Software Process Improvement (pp. 217-231). Springer, Berlin, Heidelberg.

Sacerdoti, E. D. (1974). Planning in a hierarchy of abstraction spaces. Artificial intelligence, 5(2), 115-135.

Sarraute, C., Buffet, O., & Hoffmann, J. (2012, July). POMDPs make better hackers: Accounting for uncertainty in penetration testing. In Twenty-Sixth AAAI Conference on Artificial Intelligence.

Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. arXiv preprint arXiv:1511.05952.

Schaul, T., Horgan, D., Gregor, K., & Silver, D. (2015, June). Universal value function approximators. In International conference on machine learning (pp. 1312-1320). PMLR.

Schrittwieser, J., Antonoglou, I., Hubert, T., Simonyan, K., Sifre, L., Schmitt, S., ... & Silver, D. (2020). Mastering atari, go, chess and shogi by planning with a learned model. Nature, 588(7839), 604-609.

Schölkopf, B., Locatello, F., Bauer, S., Ke, N. R., Kalchbrenner, N., Goyal, A., & Bengio, Y. (2021). Toward causal representation learning. Proceedings of the IEEE, 109(5), 612-634.

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015, June). Trust region policy optimization. In International conference on machine learning (pp. 1889-1897). PMLR.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347.

Shepperd, R., Willis, J., Hansen, E., Faber, J., Siewert, S., & Rabideau, G. (1998, March). DATA-CHASER: a demonstration of advanced mission operations technologies. In 1998 IEEE Aerospace Conference Proceedings (Cat. No. 98TH8339) (Vol. 2, pp. 419-427). IEEE.

Shivashankar, V., Alford, R., Kuter, U., & Nau, D. (2013, June). The GoDeL planning system: a more perfect union of domain-independent and hierarchical planning. In Twenty-Third International Joint Conference on Artificial Intelligence.

Silver, D., Sutton, R. S., & Müller, M. (2008, July). Sample-based learning and search with permanent and transient memories. In Proceedings of the 25th international conference on Machine learning (pp. 968-975).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. nature, 529(7587), 484-489.

Silver, D., Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., ... & Degris, T. (2017, July). The predictron: End-to-end learning and planning. In International Conference on Machine Learning (pp. 3191-3199). PMLR.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... & Hassabis, D. (2017). Mastering the game of go without human knowledge. *nature*, *550*(7676), 354-359.

Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. Journal of Artificial Intelligence Research, 39:127–177, 2010.

Simpson, W. R. (1994). Defining ATLAS 2000. IEEE Design and Test of Computers, 11, 65-65.

Siu, H. C., Pena, J. D., Chen, E., Zhou, Y., Lopez, V. J., Palko, K., ... & Allen, R. E. (2021). Evaluation of Human-AI Teams for Learned and Rule-Based Agents in Hanabi. arXiv preprint arXiv:2107.07630.

Smith, M., Donner, A., Knapp, M., Pong, C., Smith, C., Luu, J., ... & Campuzano, B. (2018). On-orbit results and lessons learned from the ASTERIA space telescope mission.

Spieker, H., Gotlieb, A., Marijan, D., & Mossige, M. (2017, July). Reinforcement learning for automatic test case prioritization and selection in continuous integration. In Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (pp. 12-22).

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., ... & Su, Z. (2017, August). Guided, stochastic model-based GUI testing of Android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (pp. 245-256).

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. ACM Sigart Bulletin, 2(4), 160-163.

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

Talamadupula, K., Smith, D. E., Cushing, W., & Kambhampati, S. (2013). A theory of intra-agent replanning. Arizona State Univ Tempe Dept of Computer Science and Engineering.

Tang, H., Houthooft, R., Foote, D., Stooke, A., Chen, X., Duan, Y., ... & Abbeel, P. (2017). # exploration: A study of count-based exploration for deep reinforcement learning. In 31st Conference on Neural Information Processing Systems (NIPS) (Vol. 30, pp. 1-18).

Tate, A. (1977, August). Generating project networks. In Proceedings of the 5th international joint conference on Artificial intelligence-Volume 2 (pp. 888-893).

Tenorth, M., & Beetz, M. (2013). KnowRob: A knowledge processing infrastructure for cognition-enabled robots. The International Journal of Robotics Research, 32(5), 566-590.

Test.ai. (2021). All Products. https://www.test.ai/all-products

Troesch, M., Mirza, F., Hughes, K., Rothstein-Dowden, A., Bocchino, R., Donner, A., ... & Campuzano, B. (2020, October). Mexec: An onboard integrated planning and execution approach for spacecraft commanding. In Workshop on Integrated Execution (IntEx)/Goal Reasoning (GR), International Conference on Automated Planning and Scheduling (ICAPS IntEx/GR 2020).

Utting, M., Pretschner, A., & Legeard, B. (2012). A taxonomy of model‐based testing approaches. Software testing, verification and reliability, 22(5), 297-312.

Van Der Krogt, R., & De Weerdt, M. (2005, June). Plan Repair as an Extension of Planning. In ICAPS (Vol. 5, pp. 161-170).

Van Hasselt, H., Guez, A., & Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

Veanes, M., Roy, P., & Campbell, C. (2006). Online testing with reinforcement learning. In Formal Approaches to Software Testing and Runtime Verification (pp. 240-253). Springer, Berlin, Heidelberg.

Vere, S. A. (1983). Planning in time: Windows and durations for activities and goals. IEEE Transactions on Pattern Analysis and Machine Intelligence, (3), 246-267.

Vinyals, O., Babuschkin, I., Czarnecki, W. M., Mathieu, M., Dudzik, A., Chung, J., ... & Silver, D. (2019). Grandmaster level in StarCraft II using multi-agent reinforcement learning. Nature, 575(7782), 350-354.

Vuong, T. A. T., & Takada, S. (2018, November). A reinforcement learning based approach to automated testing of android applications. In Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (pp. 31-37).

Waibel, M., Beetz, M., Civera, J., d'Andrea, R., Elfring, J., Galvez-Lopez, D., ... & Van De Molengraft, R. (2011). Roboearth. IEEE Robotics & Automation Magazine, 18(2), 69-82.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., & de Freitas, N. (2016). Sample efficient actor-critic with experience replay. arXiv preprint arXiv:1611.01224.

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003). PMLR.

Watkins, C. J., & Dayan, P. (1992). Q-learning. Machine learning, 8(3-4), 279-292.

Wickler, G., & Tate, A. (2012). Open On-Line Course - AI Planning. http://www.aiai.ed.ac.uk/project/plan/ooc/

Wilkins, D. E. (1990). Can AI planners solve practical problems?. Computational intelligence, 6(4), 232-246.

Willcock, C., Deiß, T., Tobies, S., Keil, S., Engler, F., & Schulz, S. (2011). An introduction to TTCN-3. John Wiley & Sons.

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8(3), 229-256.

Williams, B. G., Riedel, J. E., Bhaskaran, S., Synnott, S. P., Desai, S. D., Bollman, W. E., ... & Owen Jr, W. M. (1997). Navigation for the New Millennium: Autonomous Navigation for Deep-Space-1.

Wu, Y., Mansimov, E., Grosse, R. B., Liao, S., & Ba, J. (2017). Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation. Advances in neural information processing systems, 30, 5279-5288.

Ye, W., Liu, S., Kurutach, T., Abbeel, P., & Gao, Y. (2021). Mastering Atari Games with Limited Data. Advances in Neural Information Processing Systems, 34.

Younes, H. L., & Littman, M. L. (2004). PPDDL1. 0: The language for the probabilistic part of IPC-4. In Proc. International Planning Competition

Zheng, Y., Xie, X., Su, T., Ma, L., Hao, J., Meng, Z., ... & Fan, C. (2019, November). Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE) (pp. 772-784). IEEE.

Zheng, Y., Liu, Y., Xie, X., Liu, Y., Ma, L., Hao, J., & Liu, Y. (2021, May). Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 423-435). IEEE.

Zhu, S., Ng, I., & Chen, Z. (2019). Causal discovery with reinforcement learning. arXiv preprint arXiv:1906.04477.

Zimmerman, T., & Kambhampati, S. (2003). Learning-assisted automated planning: Looking back, taking stock, going forward. AI Magazine, 24(2), 73-73.