```
from agents.agent import Agent
import pddlgym
import networkx as nx
from construct.wrappers import PyperWrapper
from pyperplan.pddl.pddl import Type, Problem, Predicate
from pyperplan.pddl.parser import Parser
from pyperplan.planner import _ground, _search, _parse
from pyperplan.grounding import ground
from pyperplan.search.breadth_first_search import breadth_first_search
from pyperplan.search.a_star import astar_search
from pyperplan.search.a_star import greedy_best_first_search
from pyperplan.search.iterative_deepening_search import iterative_deepening_search
import random
import uuid
import copy
import networkx as nx
import click
import collections
# UPDATED to produce problem.pddl files instead of dealing with generating predicate
s etc.


class BiplexAgent(Agent):
    """
    Biplex
    """

    def __init__(self, ctx):
        super().__init__()
        self.config = ctx
        self.env = PyperWrapper(pddlgym.make(self.config['env']))
        self.env.fix_problem_index(1)
        self.env.reset()
        self.goal = self.config['goal']
        self.closed = []
        self.kg = nx.read_gml(self.config['resource_graph'])
        self.type_keyed_objects, self.token_keyed_objects = self._get_objects() # ob
jects from current state
        self.bound_objects = {}
        self.stopper = True
        self.goals = [self.goal]
        self.completed = []
        self.grounded_actions = []
        print("¿ Biplex agent initialized.\n")

    def add_objects(self, constant, typing):
        """
        Adds object to type_keyed_objects and token_keyed_objects
        """

        if constant in self.token_keyed_objects:
            return True
        try:
            self.token_keyed_objects[constant] = str(typing)
            self.type_keyed_objects[str(typing)].add(str(constant))
            return True
        except:
            raise KeyError("Not able to add to objects dicts")

        return False


    def _solve(self):
        """
        High Level Planner

        Algorithm
        1. generate a plan sketch
        2. resolve non-executable/crafting actions (by proving the objects they are
meant to create)
        - do above two steps until planner
        """
```

```
        click.secho(f"Goal: {self.goal}", fg='blue')

        # Initial planning and execution with Stripped Domain File
        status, s1 = self.sketch()
        if status:
            print("Solved.\n")
            print(f"Completed subgoals {self.completed}")
            click.secho(f"History:\n-------\n {' --> '.join(self.env.history)}\n", f
g="magenta")
            return status, s1
        return False, s1


    def sketch(self):
        """
        High-level planner operating on the stripped domain
        The stripped domain has all the craft actions, but stripped down to zero pre
cons
        """
        goal_in = self.goals.pop(0)

        s0 = self.env.observe() # Getting current state

        # We now use the non-executable domain file
        non_exec_domain_file = "agents/biplex/bias/treasure_nonexec.pddl"
        plan = []
        executable = False

        # --- Preparing problem file ------ #
        # Ground the goal
        goal = self._ground_literal(goal_in)
        click.secho(f"Goal: {goal}", fg="bright_white", bold=True)

        if goal in s0:
            self.completed.append(goal)
            return True, s0

        if goal in self.completed:
            return True, s0

        #We need to add objects from the craft actions
        # Look over nonexec domain and add objects from effects
        # we need to do this so it can come up with a plan for (have ?x-s)
        parser = Parser(non_exec_domain_file)
        domain = parser.parse_domain()
        for name, action in domain.actions.items():
            if "*" in name:
                sig = action.signature
                if len(sig) > 1:
                    raise NotImplementedError(f"Cannot handle case where action has
more than one param")
                typings = sig[0][1]
                if len(typings) > 1:
                    raise NotImplementedError(f"Cannot handle if object variable {si
g[0]} has more than one type")
                typing = typings[0]
                arg = self._ground_arg(f"{sig[0]}-{typing}")


        objects = self._get_objects_from_dicts()
        init = set(self.env.observe())
        temp_problem_file = self._generate_temp_problem_file(goal, init, objects)
        plan = self.plan(problem_file=temp_problem_file, domain_file=non_exec_domain
_file)
        executable, non_executables = self._is_plan_executable(plan)
        print(f"\tNonexc: {non_executables}")
        if plan and executable:
            print(f"\tThis is an executable plan")
            self.completed.append(goal)
            return self.execute(plan)

        if not plan:
            print(f"Current state: {self.env.observe()}")
```

```
            print(f"\tGive up. Bye.")
            return False, s0

        if not executable:
            objects_to_construct = set()
            # This means there are either non-executable actions or hypothetical act
ions
            for non_exec_action in non_executables:
                name, args = self._parse_literal(non_exec_action.name)
                for arg in args:
                    if "*" in arg:
                        if arg in self.bound_objects:## if already bound, then domai
n_file
                            continue
                        objects_to_construct.add(arg)

            for ob in objects_to_construct:
                status, s1 = self.prove(self.token_keyed_objects[ob])
                if not status:
                    print(f"Unable to construct object {ob}")
                    return False, s1
            return True, s1


        return False, s0

    def _get_objects_from_dicts(self):
        """
        Return a pddl friendly listing of objects
        """
        objects=[]
        for key, val in self.token_keyed_objects.items():
            objects.append(f"{key} - {val}")
        return objects

    def _ground_literal(self, literal):
        """
        returns a literal grounded in either objects the agent knows about OR is hyp
othesized
        """
        if self._is_grounded_literal(literal):
            return literal

        name, args = self._parse_literal(literal)
        grounded_args = []
        for arg in args:
            grounded_arg = self._ground_arg(arg)
            grounded_args.append(grounded_arg)

        return self._construct_literal(name, grounded_args)


    def _construct_literal(self, name, args):
        """
        Returns a literal based on name and args
        (have t23)
        """
        return f"({name} {' '.join(args)})"


    def _ground_arg(self, arg):
        """
        Returns a grounded arg
        NOTE: if the object is hypothetical, this is added via self.add_objects()
        """
        if self._is_grounded_arg(arg):
            return arg
        symbol, typing = self._parse_arg(arg)
        if self.type_keyed_objects[typing]:
            symbol = list(self.type_keyed_objects[typing])[0]
            return symbol
        hypo_sym = f"*{typing}_{str(uuid.uuid4())[:8]}"
        self.add_objects(hypo_sym, typing)
```

```
        return hypo_sym


    def _parse_arg(self, arg):
        """
        Returns symbol, typing
        input: ?x-t, t23-t, t23, *t234-t
        output: ?x,t  or t23,t or t23,t or *t234-t
        """

        if "?" in arg:
            if "-" in arg:
                symbol = arg.split("-")[0]
                typing = arg.split("-")[1]
                return symbol, typing
            else:
                raise ValueError(f"Argument ({arg}) Must have at least constant or t
ype")
        if "-" in arg:
            symbol = arg.split("-")[0]
            typing = arg.split("-")[1]
            return symbol, typing

        symbol = arg
        try:
            typing = self.token_keyed_objects[symbol]
            return symbol, typing
        except:
            raise ValueError(f"The object {arg} does not exist anywhere")


    def _is_grounded_literal(self, literal):
        """
        Returns true of literal is grounded
        """

        name, args = self._parse_literal(literal)
        for arg in args:
            if not self._is_grounded_arg(arg):
                return False
        return True


    def _parse_literal(self, literal):
        """
        Gets name, and arguments from a string literal as a string, list
        note: an arg could be "t23" or "?x-t" or "t23-t", assuming well formed.
        """

        name = literal.replace("(","").replace(")","").split(" ")[0]
        args = literal.replace("(","").replace(")","").split(" ")[1:]
        return name, args

    def _is_grounded_arg(self, arg):
        """
        Given an arg, checks if it is grounded
        arg = "?x-t" or "t23" or "t23-t" or "*t234"
        """

        if "?" in arg:
            return False
        return True


    def _get_objects(self):
        """
        Returns objects as a dicts, keyed by types, keyed by object
        """

        type_keyed_objects = collections.defaultdict(set)
        token_keyed_objects = collections.defaultdict()
        state = self.env.observe() ###*** LOOKS AT ENV ***********
        relevant_objects = set()
```

```
        for literal in state:
            objects_in_lit = literal.replace("(","").replace(")","").split(" ")[1:]
            relevant_objects.update(objects_in_lit)

        for o in relevant_objects:
            type_val = self.env.objects()[o]
            type_keyed_objects[str(type_val)].add(o)
            token_keyed_objects[str(o)] = str(type_val)

        return type_keyed_objects, token_keyed_objects


    def prove(self, tnode):
        # Construction
        print(f"\tExpanding resource graph for {tnode} ...")
        s0 = self.env.observe()
        try:
            actions = list(self.kg.predecessors(str(tnode)))
        except:
            print(f"\tType {tnode} does not exist in the resource graph")
            return False, s0
        if not actions:
            print("\tNode exists, but has no predecessor actions")

        while actions:
            anode = actions.pop(0)
            status, s1 = self.ground(anode)
            if status:
                return status, s1

        return False, s0

    def ground(self, anode):
        # Node expansion
        precon_types = list(self.kg.predecessors(str(anode)))
        for tnode in precon_types:
            self.goals.insert(0, f"(have ?x-{tnode})")
            status, s1 = self.sketch()
            if not status:
                return False, s1

        # once all resources have been acquired, it is time to perform craft action
        # Full fledged navigation and manipulation planning
        if anode not in self.grounded_actions:
            print(f"\tAction {anode} is grounded.")
            self.grounded_actions.append(anode)
            new_domain_file = self._create_new_domain_file(self.kg, anode, self.conf
ig['bias'])
            goal = f"(have ?x-{list(self.kg.successors(anode))[0]})"
            status, s1 = self.plan_execute(goal=goal, domain_file=new_domain_file)
            if status:
                self.goals.insert(0,self.goal)
                return self.sketch()
            return False, s1
        return True, self.env.observe()


    # Full fledged planning. Use carefully
    def plan(self, problem_file, domain_file):
        problem = _parse(domain_file, problem_file)
        task = _ground(problem)
        print(f"\tDomain: {domain_file}")
        print("\t*Planning*", end="\r")
        solution = breadth_first_search(task)
        click.secho(f"\tPlan: {solution}", fg="green")
        return solution


    def execute(self, plan):
        s0 = self.env.observe()
        if plan:
            print(f"\tExecuting actions:")
            for idx,a in enumerate(plan):
```

```
                click.secho(f"\t\t[{idx}] {a.name}", fg='red')  # need a.name here b
ecause these are Operators
                s1 = self.env.step(a.name)
            return True, s1
        print("\tNo plan found")
        return False, s0

    def plan_execute(self, goal, domain_file):
        goal = self._ground_literal(goal)
        objects = self._get_objects_from_dicts()
        init = set(self.env.observe())
        temp_problem_file = self._generate_temp_problem_file(goal, init, objects)
        plan = self.plan(problem_file=temp_problem_file, domain_file=domain_file)

        # Replace any args that have "*" with ?x-k
        new_plan = []
        to_remove = []
        for op in plan:
            name, args = self._parse_literal(op.name)
            new_args = []
            for arg in args:
                if "*" in arg:
                    # replace it with a ?x-type
                    to_remove.append(arg)
                    typing = self.token_keyed_objects[arg]
                    variable = "?x"+str(uuid.uuid4())[:3]
                    new_args.append(f"{variable}-{typing}")
                    continue
                new_args.append(arg)
            new_op = self._construct_literal(name, new_args)
            new_plan.append(new_op)

        s0 = self.env.observe()
        if new_plan:
            for idx,a in enumerate(new_plan):
                click.secho(f"\t\t[{idx}] {a}", fg='blue')
                s1 = self.env.step(a)
        else:
            return False, s0

        # Remove *k2323 from objects, and add the new k1 from (have k1) in current s
tate
        for token in to_remove:
            typing = self.token_keyed_objects[token]
            self._remove_token_from_objects(token)
            for literal in self.env.observe():
                name, args = self._parse_literal(literal)
                if "have" in name:
                    type_of_arg = self.env.objects()[args[0]]
                    if type_of_arg.name == typing:
                        self.add_objects(args[0],typing)

        self.completed.append(goal)
        return True, s1

    def _remove_token_from_objects(self, token):
        typing = self.token_keyed_objects[token]
        del self.token_keyed_objects[token]
        self.type_keyed_objects[typing].remove(token)
        return True


    def _generate_temp_problem_file(self, goal, init, objects):
        goal_line = f"(:goal {goal})"
        init_line = "(:init" + " " + " ".join(init) + ")"
        objects_line = "(:objects" + " " + " ".join(objects) + ")"
        domain_line = "(:domain treasure)" #HACK TODO need to fix
        define_line = "(define (problem treasure)" #HACK TODO fix

        self.problem_file = "agents/biplex/temp/problem_gen.pddl"
        with open(self.problem_file, "w+") as f:
            f.write(define_line)
            f.write("\n\n")
```

```
            f.write(domain_line)
            f.write("\n\n")
            f.write(objects_line)
            f.write("\n\n")
            f.write(init_line)
            f.write("\n\n")
            f.write(goal_line)
            f.write("\n\n")
            f.write(")")
        return self.problem_file


    def _is_plan_executable(self, plan):
        """
        Returns False if any part of the action literal contains a "*"
        (*craftk *k_b23132)
        (*crafk k2)
        (pickup *k_b242323)

        Also returns the action literal
        """
        nonexec_actions = []
        flag = True
        if not plan:
            return False, []
        for action in plan:
            if "*" in action.name:  #actually this is an Operator in Pyperplan repre
sentation
                flag = False
                nonexec_actions.append(action)
        if flag:
            return True, []
        return False, nonexec_actions


    def _create_new_domain_file(self, graph, anode, current_domain):
        action_symbol = f"\t(:action {anode}\n"
        precon_types = list(graph.predecessors(str(anode)))
        precons = []
        effects = []
        params = []
        for p in precon_types:
            variable = "?x"+str(uuid.uuid4())[:3]
            param = f"{variable} - {p}"
            pred = f"(have {variable})"
            effp = f"(not {pred})"
            precons.append(pred)
            effects.append(effp)
            params.append(param)

        eff_types = list(graph.successors(str(anode)))
        for e in eff_types:
            variable = "?y"+str(uuid.uuid4())[:3]
            param = f"{variable} - {e}"
            effp = f"(have {variable})"
            effects.append(effp)
            params.insert(0,param)

        param_line = f"\t\t:parameters ({' '.join(params)})\n"
        precon_line = f"\t\t:precondition (and {' '.join(precons)})\n"
        effects_line = f"\t\t:effect (and {' '.join(effects)}))"

        action_entry = action_symbol+param_line+precon_line+effects_line


        new_domain_filename = current_domain.split(".pddl")[0]+"_gen.pddl"
        with open(current_domain,'r') as current, open(new_domain_filename,'w') as s
econdfile:
            lines = current.readlines()
            for line in lines[:-1]:
                secondfile.write(line)
            secondfile.write("\n")
            secondfile.write(action_entry)
```

```
            secondfile.write("\n")
            secondfile.write(")")

        return new_domain_filename
```