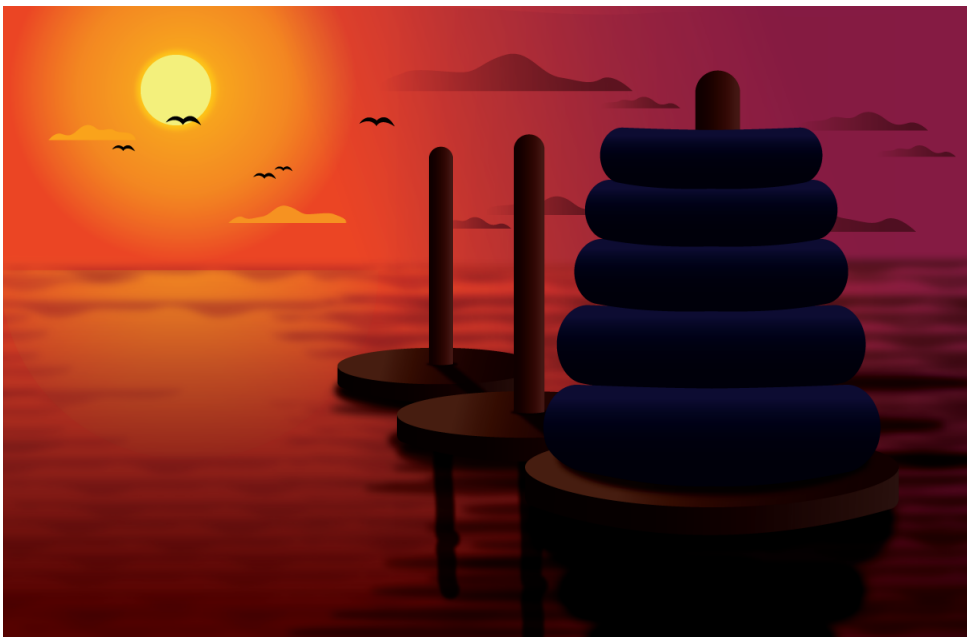
Introduction to Artificial Intelligence
Exercise 4: GraphPlan



Deadline: July 11 2024

1 Introduction

In this project, you will implement parts of the Graphplan algorithm and design heuristics that are derived from the planning graph. In the first part, you complete the implementation of the Graphplan and test the algorithms on problems from the "dock-worker robot" domain. In the second part, you will use a relaxed version of the planning graph to derive heuristics for A*. In the last part, you will automatically create domain and problem files for the Tower of Hanoi puzzle.

The code for this project consists of several Python files, some of which you will need to read and understand in order to complete the assignment, and some of which you can ignore. You can download all the code and supporting files (including this description) as a zip archive.

Files you'll edit:

`graph_plan.py` - Where the graphPlan algorithm runs, this module in charge of creating the Graphplan, extending it if needed and extracting a plan.

`plan_graph_level.py` - Representation of one level (actions layer and propositions layer) of the graph.

`planning_problem.py` - Representation of planning problem as a search problem.

`hanoi.py` - Where the domain and problem files are created.

Files you might want to look at:

`action.py` - The action object.

`proposition.py` - The proposition object.

`action_layer.py` - The ActionLayer object.

`proposition_layer.py` - The PropositionLayer object.

`util.py` - Useful data structures.

What to submit:

Code - You will fill in portions of graph_plan.py, plan_graph_level.py, planning_problem.py and hanoi.py during the assignment. In addition, you will create two instances of the DWR domain. These problem files should be named `dwr1.txt` and `dwr2.txt`, respectively. You should submit these six files and the `README.txt`.

Answers - You will answer question that will require you to think about the theory, your results and what's between them. These questions are marked by "Understanding Question". You should submit a pdf file with your answers.

Each team should submit exactly one tar file that contains all files from above.

Evaluation: Your code will be autograded for technical correctness. The autograder machine is running Python 3 (don't use Python 2.7). Please do not change the names of any provided functions or classes within the code, or you will wreak havoc on the autograder. Please make sure you follow the `README` format **exactly**.

Academic Dishonesty: We will be checking your code against other submissions in the class for logical redundancy. If you copy code from someone else and submit it with minor changes, *we will know*. These cheat detectors are quite hard to fool, so please don't try. We trust you all to submit your own work only; *please* don't let us down. If you do, we will pursue the strongest consequences available to us.

Getting Help: If you have any question, you are probably not alone. Please post your questions via the relevant exercise forum on the course Moodle. **Please do not write to our personal e-mail addresses!**

README format: Please submit a `README.txt` file. The `README` should include the following lines (exactly):

1. id1 --- student 1 id
2. id2 --- student 2 id
3. ***** --- 5 stars denote end of i.d information
4. comments

For an example check out the README.txt provided with your project. This `README` will be read by a script, calling the autograder. Note that if you decide to submit alone, you should remove line 2, i.e.:

1. id1 --- student 1 id
2. ***** --- 5 stars denote end of i.d information
3. comments

Ignore the hyperlinks to the code files in this PDF. Instead see the code files we provided.

2 GraphPlan

We will start with completing all the necessary methods for the GraphPlan algorithm. Our algorithm will build the planning graph with mutex relations - each level in the planing graph is composed of two layers: the actions layer and the propositions layer. First, we will complete the method that detects mutex relations between actions and propositions.

Question 1 (2 points)

Two actions at the same level are mutex if one of the following is satisfied: they have inconsistent effects, they interfere with each other or they have competing needs. Two of those conditions can happen regardless of the other actions and propositions in the level, so it will be a good idea to check these conditions before the creation of the graph. Implement the `independent_pair` function in `graph_plan.py`. This function returns true only if the actions don't have inconsistent effects and don't interfere with one anoher.

Hint: Make sure to check out `action.py`!

Question 2 (2 points)

We will now complete the formulation of mutex actions; implement the `have_competing_needs` function in `plan_graph_level.py`. This function returns true if the two actions have competing needs, given the list of the mutex propositions from the previous level (list of pairs of propositions).

Hint: Check out `Pair` in `util.py`.

After you have implemented `have_competing_needs`, go over the function `mutex_actions`. This function returns true if the actions `a1` and `a2` are mutex actions (given the list of the mutex propositions from the previous level). We first check whether `a1` and `a2` are in `PlanGraphLevel.independent_actions`, which is the list of all the independent pair of actions (according to your implementation in question 1). If so, we check whether `a1` and `a2` have competing needs.

Question 3 (2 points)

Implement the `mutex_propositions` function in `plan_graph_level.py`. This function returns true if the two propositions are mutex, given the list the of mutex actions form the current level (list of pairs of actions).

Hint: Remember that two propositions are mutex, if all ways of achieving the propositions (that is, actions at the same level) are pairwise mutex.

Hint: `Proposition.get_producers` returns the list of all the possible actions in the layer that have the proposition on their add list.

The following 5 questions complete the implementation of the GraphPlan algorithm. After we have detected mutex relations we can expand the planning graph; using the previous proposition layer we can construct the current actions layer.

Hint: Make sure to check out `action.py`, `proposition.py`, `action_layer.py` and `proposition_layer.py`.

Question 4 (2 points)

Implement the `update_action_layer` function in `plan_graph_level.py`. This function receives the previous proposition layer (go over `action_layer.py` and `proposition_layer.py`!) and updates the actions in the current action layer.

Hint: We add an action to the layer if its preconditions are in the previous propositions layer, and the preconditions are not pairwise mutex.

Question 5 (1 points)

Implement the `update_mutex_actions` function in `plan_graph_level.py`. This function receives a list of the mutex proposition in the previous level and updates the mutex actions in the current action layer.

Next, Using the current actions layer we construct the current propositions layer.

Question 6 (2 points)

Implement the `update_proposition_layer` function in `plan_graph_level.py`. This function updates the propositions in the current proposition layer, given the current action layer (`self.action_layer`).

When you add a proposition to the current layer, don't forget to update the producers list!

Hint: The same proposition in different layers might have different producer lists, thus two different instances should be created.

Hint: Go over `proposition.py`!

Question 7 (1 points)

Implement the `update_mutex_proposition` function in `plan_graph_level.py`. This function updates the mutex propositions in the current proposition layer.

Hint: We saw in the tirgul that there are two types of mutex relation between propositions. however, only one of them is relevant when we use STRIPS to represent the planning problem.

Now we can complete the expansion of the planning graph.

Question 8 (1 points)

Implement the `expand` function in `plan_graph_level.py`. This function receives the previous level and updates the current. Your algorithm should work as follows: first, given the propositions and the list of mutex propositions from the previous layer, set the actions in the action layer. Then, set the mutex action in the action layer. Finally, given all the actions in the current layer, set the propositions and their mutex relation in the propositions layer.

Now you should be able to run your code on the provided domain and problem:

```
python3 graph_plan.py dwrDomain.txt dwrProblem.txt
```

This domain and problem are simplifications of the dock-worker-robot domain (for which you can find the full specification here). In this simplified domain, there are two robots `q` and `r`, two containers `a` and `b` and two locations `1` and `2`. Each robot and container can be in either location (e.g., the proposition `r2` represents the fact that the robot `r` is at location `2`). In addition, each robot can holds at most 1 container (e.g., the proposition `uq` represents the fact that the robot `q` is free and the proposition `bq` represents the fact that the robot `q` holds at most the container `b`). The robots can move between the two locations. This simplified domain will make debugging easier, and it has already been propositionalized so that you can directly apply GraphPlan.

Hint: The solution found by your implementation should return a plan with 6 actions (excluding 4 `noOp` actions).

Question 9 (2 points)

Create two more problem instances in the DWR domain by changing the initial state or the goal state or both. One instance (`dwr1.txt`) should have a goal state that can be achieved within at least 8 actions (not including `noOps`). The other instance (`dwr2.txt`) should have a goal state that cannot be achieved; make sure your code fails on this problem.

```
python3 graph_plan.py dwrDomain.txt dwr1.txt
python3 graph_plan.py dwrDomain.txt dwr2.txt
```

3 Planning Graph for heuristics for A*

An effective approach to planning is to derive heuristics from the planning graph and then to use a search algorithm for choosing operators and to generate a plan.

Question 10 (2 points)

Complete the implementation of `PlanningProblem` in `planning_problem.py` as a search problem.

Note: A state must be hashable! Therefore, you might want to represent a state as a frozenset.

In order to run the search algorithms, you can either add your `search.py` from project 1 to the project folder, or use the `search.py` file we provided. If you choose to use our file, there is no need to move any file. Now, your search agent should solve:

```
python3 planning_problem.py dwrDomain.txt dwrProblem.txt zero
```

where zero means the null heuristic.

Hint: Is it possible that a `noOp` action will be in an optimal plan?

Question 11 (2 points)

Implement the `max_level` heuristic in `planning_problem.py`. The heuristic is computed as follows: for each state, expand the planning graph, omitting the computation of mutex relations, until you reach a level that includes all goal propositions. The heuristic value is the number of levels required to expand all goal propositions. If the goal is not reachable from the state your heuristic should return `float('inf')`.

Hint: The expansion of the planning graph in the heuristic calculation is very similar to the one in question 8. You can, but you don't have to, use the methods that you have already implemented in `plan_graph_level.py` and implement part of the heuristic in `expand_without_mutex`.

Hint: `is_fixed` returns true if the graph hasn't changed in the last expansion.

Hint: It might be a good idea to check out `graph_plan` in `graph_plan.py`

Now, your search agent should solve:

```
python3 planning_problem.py dwrDomain.txt dwrProblem.txt max
```

Question 12 (2 points)

Implement the `level_sum` heuristic in `planning_problem.py`. This heuristic is computed as follows: for each state, expand the planning graph, omitting the computation of mutex relations, until you reach a level that includes all goal propositions. The heuristic value is the sum of the sub-goal's level where they first appeared. If the goal is not reachable from a state your heuristic should return `float('inf')`.

```
python3 planning_problem.py dwrDomain.txt dwrProblem.txt sum
```

Question 13 - max level and level sum (3 points) - Understanding Question

In this question we will compare different heuristics we can derive from a planning graph, theoretically and empirically. We will focus on the max level and level sum heuristics.

Note: In this question we talk about the case where we use forward A* tree search to solve the planning problem, guided by a heuristic derived from the planning graph. In particular, here when we talk about optimality we refer to the question of whether the search algorithm finds the shortest plan possible.

Note: In all the questions, if you claim a property of an heuristic explain why this property holds, even if we stated it in class. We expect explanations, no formal proofs required.

Optimality

1. Theoretically, for each of the heuristics - is its optimality guaranteed?
2. Empirically, what are the lengths of the plans you found for the DWR problem (in questions 11 and 12) with each of the heuristics? Include also the null heuristic results in your comparison. For each of these - is it an optimal plan?

3. Are the theoretical and empirical results consistent with each other? If so, explain. If not, explain how this is possible.

Running Time

1. Theoretically, can we claim that one of the heuristics is guaranteed to expand less-or-equal nodes than the other heuristic (in the general case)?
2. Empirically, how many search nodes were expanded with each one of the heuristics? So which one was more efficient in this case?

Question 14 - set level (3 points) - Understanding Question

In this question we will consider theoretically a heuristic you do not implement in the exercise - the set level heuristic.

Reminder: the set-level heuristic returns the level at which all the propositions in the goal appear in the planning graph without mutex between any pair of them.

Just like in the previous question, we will consider the case of using this heuristic to guide an A* search.

1. Is the optimality of this heuristic (in the same sense as in the previous question) guaranteed?
2. What is the relation between this heuristic and the max level heuristic in terms of number of nodes expanded?
3. Is this heuristic perfect in the sense that it always returns the precise distance to the goal?

4 Tower of Hanoi

The Tower of Hanoi consists of three pegs and a number of disks of different sizes. The puzzle starts with the disks in a neat stack in ascending order (by size) on one peg, the smallest at the top, thus making a conical shape. The objective of the puzzle is to move the entire stack to another peg, obeying the following constraints:

- Only one disk can be moved at a time
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack
- No disk may be placed on top of a smaller disk

The goal of the this section is to automatically create domain and problem files for the Tower of Hanoi problem for any number of disks and pegs. For any `n` and `m` we will enumerate the disks from `0` to `n-1` where `0` is the smallest disk and `n-1` is the largest, and we will enumerate the pegs from `0` to `m-1`.

Question 15 (3 points)

Implement the `create_domain_file` function in `hanoi.py`. This function receives as input: the string `domain_file_name` and two integers `n` and `m`. The function creates the domain file (named '`hanoi_n_m_domain.txt`') for the the Tower of Hanoi puzzle with `n` disks and `m` pegs. See `dwrDomain.txt` for an example of a domain file.

Question 16 (2 points)

Implement the `create_problem_file` function in `hanoi.py`. The function receives as input: the string `problem_file_name` and an integer `n`. The function creates the problem file (named `problem_file_name`) for the the Tower of Hanoi puzzle with `n` disks and `m` pegs. In the initial state, all the disks are on the first peg (i.e. `p_0`) in a neat stack in ascending order of size (i.e., disk `n-1` at the bottom). In the goal state all the disks are in the same order but on the last peg (`p_(m-1)`). See `dwrProblem.txt` for an example of a problem file.

Now, for every positive integers `n` and `m` the command:

```
python3 hanoi.py [n] [m]
```

Should create the files `hanoi_[n]_[m]_domain.txt` and `hanoi_[n]_[m]_problem.txt` (for example, the command `python3 hanoi.py 3 3` should create the files `hanoi_3_3_domain.txt` and `hanoi_3_3_problem.txt`)

Hint: The minimum number of moves required to solve a Tower of Hanoi puzzle with 3 pegs is $2^n - 1$.

Good Luck!