IJCAI–15 Experiment Rationale and Design

Dustin Dannenhauer
Lehigh University
Bethlehem, PA USA
dtd212@lehigh.edu

Abstract

The following is a rough outline of this document. (1) Why are richer expectations are needed? (2) A high level english description of how to compute these expectation. (3) Comparison of this new technique against other GDA research. (4) Proposed experimental setup to show how this new technique compares to previous methods of obtaining expectations and checking they are met.

1 Motivation for richer expectations

Previous GDA systems such as ARTUE (Molineaux and Aha ACS 2014) represent expectations as an instance of a state. In their ACS 2014 paper: "The planner outputs (1) a plan π_c, which is an action sequence $A_c = [a_{c+1}, ..., a_{c+n}]$ and (2) a sequence of expectations $X_c = [x_{c+1}, ..., x_{c+n}]$, where $x_i \in X_c$ is the state expected after executing a_i in A_c and $x_{c+n} \in g_c$." This is representative of most GDA systems[1]. Expectations are the state after the action has been executed and are computed at the time of plan generation by keeping track of the state after each action has been added to the plan.

Discrepancy detection (discrete only) compares expectations to the observed state and an anomaly occurs if either some fact in the observed state is not in the expected state or the observed state is missing a fact contained in the expectation. In Molineaux's work this is sufficient because they test their system on a small domain with small states[2] and expectations are only meant to describe the expected change in state after a single action. As we will show later in the paper, more real-world situations will require agents to have expectations for executing more than 1 action.

Expectations are meant to capture the required change(s) that an action should have on the environment. By representing expectations as states unnecessary information is held by the expectation in all but the simplest domains (i.e. MudWorld is a simple enough domain). An example of a domain where expectations cannot be represented as entire state is Starcraft, the real-time strategy game[3]. Thus we come to the question: "What information should an expectation contain?".

2 Algorithm High-Level Description

The natural first answer to this question: "What information should an expectation contain?" is to use effects of the operator that produces the action as it is added to the plan[4]. However, at some point in time, those facts may then be removed by other operators as part of their delete list, in which case we should remove them from the expectation. Thus our planner will output a sequence of expectations $X_c = [x_{c+1}, ..., x_{c+n}]$, where $x_i \in X_c$ is a set of facts such that $x_i \subset xs_i$ where xs_i is the expected state after executing a_i in A_c. Thus this is the primary motivation for our algorithm, which calculates expectations in this manner.

3 Background

Describe all other GDA research regarding expectations.

4 Proposed Experiment

I need to show expectations

4.1 Domain

The domain will be a simple tile world inspired by *MudWorld* (Molineaux and Aha - ACS 2014). This domain will be catered toward two types of tasks the agent will perform: navigation tasks and perimeter tasks. Tiles in the world will have a 40% chance of being mud, which are inhibitors of navigation and perimeter tasks (because both involve movement). If a tile is a mud tile then the agent will become stuck and must generate the goal to become unstuck before resuming its previous goal. The domain will also have another type of obstacle like mud, but something that won't affect navigation tasks, only perimeter tasks.

Goals

- *Travel from start to destination (expectations: no mud) - mud causes rover to be stuck, rover needs to perform unstuck manueaver.
- (Idea 1) Travel from start to destination to start (exploration mission) and place battery charging stations along the way so that the rover can make its way back (suppose the rover can only carry one battery at a time).
- (Idea 2) Place 3 battery charging stations at given sites (expectations: clear skies, i.e. no trees - clouds are events that prevent this?!)

- *(Idea 3) Place beacons to signify good landing sites for incoming spacecraft - lets say magnetic radiation disturbs the ability for the beacon to transmit its location. The beacon both transmits signals to the agent and the incoming spacecraft. Expectation is that beacon signal either doesn't make it to the agent or is incorrect (altered) when the agent gets it. Thus this situation does not affect navigation tasks only perimeter tasks.

Planner

The planner will be python SHOP. The grounded variables in the state will include the rover, each tile (TileA1, TileC3, etc) The domain model will have the following operators and methods:

- Operator for moving rover north, south, west, east
- Method for moving from one tile to another tile not adjacent
- Method for getting unstuck (e.g. clean wheels)
- Operator for performing perimeter related actions

4.2 Measuring Performance

The following data will be collected during experiments:

- Number of plans achieved (recall)
- Number of plans failed (precision)
- Number of anomalies correctly identified
- Number of anomalies missed (went undetected)
- Number of false positive anomalies (no anomaly but system detected anomaly)

4.3 Agents

1. **OperatorEffects** This system will generate expectations that are simply the effects of the operator to be performed. **Prediction of behavior:** This system will not identify obstacles prohibiting perimeter goals and will thus fail on all instances when an obstacle inhibiting a perimeter goal occurs.
2. **MolineauxAha2014** The baseline system will use expectations that are states, just like in (Molineaux and Aha - ACS 2014). The expectations will be generated at the same time the plans are generated and will be checked after each action is performed by the agent. **Prediction of behavior:** The agent's descrepancy detection will incorrectly identify expectations for plans achieving navigation goals because of the facts in the state that are obstacles to the perimeter goals. These obstacles shouldn't affect the navigation goals but because of the way these expectations are generated they will incorrectly identify anomalies [5].
3. **HTNLearnedExpectations** This system will use our LearnExpectations Algorithm to generate expectations at the time of planning and will correctly detect all discrepancies and will not detect any false-positive descrepancies.

5 Summary of Molineaux's Experiments:

Primary Question: How exactly did learning new event models cause the agent to reach its destination faster? Did the planner create a new plan? No, because it did not know where mud would be and therefore could not plan around it. Considering this, did the system generate a new goal when it saw a mud tile, and the new goal resulted in a plan to take a different route avoiding the mud tile? I found this sentence: "...execution cost is lower in each domain when planning with knowledge of the unknown event." The agent's in Molineaux's paper do not have goal formulation or goal management components (end of 3.1). After rereading the paper i think only in plan generation are the new modelss learned, this is because of the last section of 5.1. After discussing this with Hector, I've come to the conclusion that what their system did is create a plan and execute it. When the system came to a mud tile it would just go through it treating it like any other tile. Then it would be slowed and arrive at the next tile later than expected. This would generate a discrepancy and then an explanation would occur. The system would then learn that going through a mud tile causes it to slow down. Therefore the planner would then create a plan and expectations such that it expects to not go through any mud tiles. Therefore when the agent does get to a mud tile the planner would replan from that point taking in the mud tile into account, and thus would reach the destinatoin faster avoiding any possible mud tiles.

- *FoolMeTwice* was evaluated on the execution cost to achieve its goals. Execution cost is the "time taken to achieve the goal". **Hypothesis:** After learning unknown event models, *FoolMeTwice* will create plans that require less time (*to execute*). There are no explicit learning goals
- *FoolMeTwice* was tested in two domains:
  - *Satellites* (based on the IPC 2003). For now I do not focus on this domain.
    - Each scenario has 5 goals, requiring that an image of a random target be obtained in a random spectrum
  - *MudWorld* (similar to Mars Rover domain from Molineaux's earlier work)
    - 6x6 grid, random start and destination locations
    - Every tile has a 40% chance of having mud
    - All routes between start and end are at least 4 steps
    - (Assumed from paper) Every scenario only has one navigation goal
- Each domain had randomly generated 50 training scenarios and 25 test scenarios. Experiments were run by doing 5 rounds of: train on 5 training scenarios, run on all test scenarios, record data point, train on a different 5 training scenarios, run all on all test scenarios, record data point, rinse and repeat until you have 5 data points. Now replicate the whole process 10 times, and average all the first data points, all the second data points, etc.

6 Experiments

6.1 Domain

The domain will be a square grid of size n (for now choose n=6) that will have mud tiles being generated with a probability of 40% and magnetic radiation clouds (unobservable by the agent) occuring in a spot with 30% probability that will last 3 ticks (i.e. one tick is the time it takes to execute one action). The goals of the agent will be either be to navigate from one location to another or mark multiple sites with a beacon. Mud is the only obstacle for navigation and magnetic radiation is the only thing that disrupts beacons. If magnetic radiation occurs in the same location as where a beacon is placed, the agent will lose communication with that beacon and thus assume it isn't there.

Experiments will be run in the following manner:

1. Generate $X + Y$ goals, where X is a number of mud goals and Y is a number of beacon goals
2. Run each agent in the domain with the same goals given in the same order (assume a user is giving the goals one at a time)
3. Record data from experiments and compare agents

If expectations are not violated during plan execution, then plan is considered success.

6.2 Measuring Performance

The following data will be collected during experiments:

- Number of plans achieved
- Number of plans failed (accumulation of the following three data categories)
- Number of anomalies correctly identified
- Number of anomalies missed (went undetected)
- Number of false positive anomalies (no anomaly but system detected anomaly)

6.3 Agent Implementations

Each agent will be given one goal at a time. That agent will generate a plan using the PyHOP planner. Then the plan will be simulated by creating a state variable starting with the current state of the environment and applying actions. As each action is being applied to the current state, there is a 10% that any given tile will have a magnetic cloud appear on it and will last for 3 ticks (may increase from 3 later). After the action is applied to the state, the agent checks its expectations against its observations. If expectations match and it's the last action in the plan, plan is achieved. Otherwise if expectations don't match, plan is considered failed and we record data about the expectations (false positive, correctly identified). If plan fails (need ground truth checking here) but expectations say it did succeed, record this as anomaly missed.

Each agent will be the same accept that we will simply switch out the expectations of that agent. Each agent will have different expectations output to the discrepancy detector.

6.4 Coding Requirements/Flow

1. Create PyHOP domain files for mud world
2. Test PyHOP on ability to produce correct plans for both nav and beacon goals
3. Build simulator which will take some state and action, and return the new state which is the result of applying that action on that state
4. Create three different expectation generation functions, one for each agent
5. Test expectations are working
6. Test each agent in mud world simulator with expectations and events like magnetic clouds appearing
7. Code experiment setup involving (an agent executing multiple goals, recording when plans fail or succeed and when expectations are missed, falsely detected, or correct)
8. Graph data from experiments

7 Another Experiment

see footnote 5

8 Important things for later

1. Need to discuss discrepancy detection and how it is dependent on representation of expectations.
2. Need to discuss how expectations are dependent on state representation.
3. Making the problem harder (numerical constraints, i.e. probabilistic expectations?)
4. Suppose expectations are just a full state and discrepancy detection is able to know what atoms in the expectation are relevant to compare to the current state, where does it get that knowledge? This is not a valid counter example.

[1] literature review needed to confirm this

[2] would be nice to quantivately describe the size of the states

[3] are there any references to how big a real-world domain would be for a robot?

[4] I am assuming the planner is an HTN planner throughout this document

[5] There is an interesting discussion to be had here. One may be able to argue for a simpler expectation generation algorithm and just expect to generate false-positives of anomalies and rely on the explanation generation to realize these anomalies are unfounded. But if explanation is expensive or knowledge-intensive, then it pays to have better expectations. We could run a further experiment testing this behavior.