FDRTool - DeltaMax and AlphaStar implementation

Boulionis Athanasios 189

Kolyda Maria 172

Fact.h/Fact.cpp:

What is a **Fact**?:

A Fact represents a variable-value pair, like a statement that a specific variable has a specific value.
Example: If you have a variable `Color` with possible values `Red`, `Blue`, a Fact could be `Color=Red`.

Structure of the Fact Class:

1. *Variable *var:*
   A pointer to a Variable object (e.g., Color, Temperature).
   The Variable defines the domain (possible values) for this fact.
2. *Object *val:*
   A pointer to an Object object (e.g., Red, 25°C).
   The Object is the current value of the variable.

Methods:

1. *setVariable(Variable *v) / getVariable():*
   Assign or retrieve the variable associated with this fact.

```
Variable *myVar = new Variable("Color");
Fact fact;
```

```
fact.setVariable(myVar); // Fact now represents "Color=?"
```

2. *setValue(Object *v) / getValue():*

   Assign or retrieve the value of the variable.

```
Object *red = new Object("Red");
fact.setValue(red); // Fact now represents "Color=Red"
```

3. *operator==:*

   Checks if two facts are equal by comparing their variables and values.

   Example:

   `Color=Red` == `Color=Red` → `true`

   `Color=Red` == `Temperature=25°C` → `false`.

4. *toString():*
   Returns a human-readable string like "VariableName=Value".
5. *getNumber():*

   Returns the index of the fact's value in the variable's domain (e.g., Red might be index 0).

6. *getDomainSize():*

   Returns the total number of possible values for the variable (e.g., Color might have 3 possible values: Red, Green, Blue).

DeltaMax Algorithm Implementation

For the implementation we used the following algorithm (shown in pseudocode):

```
DeltaMax(s):
   For each p do  // in problem not s
      if p ∈ s then
         Δmax(s,p) ← 0
      else
         Δmax(s,p) ← ∞
   U ← s

   Iterate
      For each a such that pre(a) ⊆ U do
      U ← U ∪ eff(a)
    For each p ∈ eff(a) do
      Δmax(s,p) ← min{Δmax(s,p), cost(a) + max{Δmax(s,q) | q ∈ pre(a)}}
   Until no change occurs in the above updates
End()
```

The purpose of DeltaMax heuristic function is to estimate the “cost” to reach a goal state from the current state, under a relaxed planning assumption; preconditions of actions only require that all their facts are reachable - there is no deletion of facts.

First we initialize the holder variables:

```
unordered_map<String, int> deltaMax;
set<string> U;
```

`deltaMax`: Acts like dictionary (in python) or a hash map (in java), it’s a matrix that holds <key, value> pairs. We use it to assign the “deltamax” cost of each `fact` in the problem.

`U`: Stores states that are already known to be reachable from the current state.

```
// Get all possible facts and initialize them infinity
for (Fact& fact : vars) {
    Variable* var = fact.getVariable();
    Domain domain = var→getDomain();
    for (int i=0; i<domain.getSize(); i++) {
        Object value = domain.getValue(i);
        string fact_str = var→getName() + "=" + value.getDescription();
        deltaMax[fact_str] = std::numeric_limits<int>::max();
    }
}

// Set reachable states to 0
for (Fact& fact : vars) {
    string fact_str = fact.toString();
    deltaMax[fact_str] = 0;
    U.insert(fact_str);
}
```

In this loop we fill `dealtaMax` with the initial values. We assign the delta cost to each of all the possible facts.

- `<infinity>` → Fact is unreachable from the current state.
- `0` → Fact exists in the current state. Mark as reachable by inserting it into `U`.

```
bool updated;
    do {
    updated = false;
    ...
    } while (updated);
```

At every pass, we iterate through all the possible action that can be taken from the current state and search for lower costs or new facts. If during that pass any change occurs, we repeat a pass through the loop, until no more changes happen, in which point we stop.

Inside this loop:

```
    for (Action& action : actions) {
        // Get all valid parameter combinations for each action
        auto allParams = action.isApplicable(rigids, vars);
        ...
```

For every action, we get all the valid parameter combinations. The `isApplicable(...)` method returns a list of all tuples of objects (e.g. `("blockA", "table")`, `("locationA", "dock_1")`) that could fill this action, even before checking preconditions.

```
for (auto& params : allParams) {
            GroundedAction ga(&action, params);
            std::vector<Fact> preconditions = ga.getPreconditions();
            ...
```

Then, we take each of these tuples and turn them into a “grounded” action. A grounded action (ga) is the concrete action with specified parameters - for example, “move `blockA` to `table`” - so we can inspect exactly the preconditions needed and the effects of this action on the current state.

We extract the list of precondition facts that must be true in order for this specific grounded action to be applied.

```
// Check if preconditions are satisfied
            bool precsMet = true;
            int maxPreconditionCost = 0;
            for (const Fact& prec : preconditions) {
                string precStr = prec.toString();
                if (!U.count(precStr)) {
                    precsMet = false;
                    break;
```

```
            }
            maxPreconditionCost = std::max(maxPreconditionCost, deltaMax
[precStr]); // Find the most expensive precondition to execute this action
        }
                ...
```

Then, we check if the preconditions are met, by checking if they exist inside `U` and among the reachable preconditions, we record the highest cost (an action can only fire once *all* its preconditions are in place, so it must wait for the slowest ones).

```
        if (precsMet) {
            int actionCost = action.getCost();

            for (int i=0; i < ga.getEffectsCount(); i++) {
                Fact eff = ga.getEffect(i);
                string effStr = eff.toString();

                int newCost = actionCost + maxPreconditionCost;

                if (!deltaMax.count(effStr)) {
                    deltaMax[effStr] = std::numeric_limits<int>::max();
                }

                if (newCost < deltaMax[effStr]) {
                    deltaMax[effStr] = newCost;
                    U.insert(effStr);
                    updated = true;
                }
            }
        }
```

Once all preconditions are met, we retrieve `actionCost` which is the cost of executing a particular action.

Then, we iterate through each effect of the ground action, compute the new cost `newCost` = cost to satisfy all preconditions (so, the slowest one) + the action's own cost.

Then, if `newCost` improves upon the stored cost, update `deltaMax[effStr]`, add the fact to `U`, and set `updated = true` to do another pass through the loop.

This should yield an admissible deltaMax heuristic, to guide search in planning problems!

Algorithm selection logic

At runtime, the program prompts the user to select the search algorithm:

```
cout << "Select algorithm(1 for BFS, 2 for A*) : 1. BFS 2. A* ";
    int choice;
cin >> choice;

FDRSTate *r ;

if (choice == 1) {
    r = BFS2(init,goal,examined,mem);
} else if (choice == 2) {
    r = Astar(init, goal,examined,mem);
}
```

If the user enters 1,
Breadth-First Search (BFS) is executed. Otherwise, if the user enters 2, **A*** with the Max Cost heuristic is used. This allows easy comparison between BFS and heuristic (A*) search strategies within the same planning framework.

Experimental results

To integrate the Max Cost heuristic into the FDRTool, the following change was made.

```
if (!deltaMax.count(effStr)) {
    std::cerr << "[WARN] Skipping unknown effect: " << effStr << std::endl;
    continue;
}
```

Runtime protection: Added a check to avoid segmentation faults when the heuristic attempted to access unknown facts.

To evaluate the implementation, various test cases were used. The tests were performed using both the BFS algorithm and A* with the Max Cost heuristic function.

Case	Algorithm	Time (s)	States	Memory
blocks0.fdr	BFS	6.792	35	57
blocks0.fdr	A*	1.687	111	128
blocks1.fdr	BFS	2.77	842	866
blocks1.fdr	A*	1.766	236	519
blocks2.fdr	BFS	3.717	677	800
blocks2.fdr	A*	3.208	236	519
blocks3.fdr	BFS	11.09	4373	5932
blocks3.fdr	A*	29.137	6924	9124
blocks3B.fdr	BFS	19.383	5232	16328
blocks3B.fdr	A*	97.122	15318	26979
gripper0.fdr	BFS	0.747	94	165
gripper0.fdr	A*	1.637	38	130
gripper1.fdr	BFS	1.241	62	98
gripper1.fdr	A*	0.738	56	137

A* proved to be faster and explored fewer states, while also consuming less memory in many cases.

The Max Cost heuristic significantly improves performance in problems with well-defined goals , reducing both the number of states and execution time.

However, in more complex problems with multiple branches or fewer constraints (e.g., the *gripper*), its effectiveness may decline, and it may consume more resources than BFS.