Programming Assignment 4

Aditya Gupta
2015CSB1003

April 7, 2017

1 Planners for Block World

1.1 Introduction

The goal of this lab was to implement a planning agent for solving Block World Problem wherein given N blocks and actions to "pick a block from table", "unstack a block from another block", "release a holding block" and "stack a holding block onto another block" we were required to reach a goal state configuration from an initial state using three different planners, "forward search planner" with BFS and A*; "goal stack planning". The initial and final state is composed of propositions such as `(on 1 2)` and `(ontable 3)`. Each action is a transformation of these propositions which has some `preconditions`, which when true allows us to perform that action and the `effects` of that actions are unified with the current state and the negative literals removed. States are represented in PDDL.

1.2 Implementation Details

1.2.1 Proposition

Each predicate was assigned a constant value(f(pre) for predicate `pre`) such that 1 for '`on`' till 5 for '`empty`', also the negation of these predicates were represented as negative of the corresponding value. Then the further arguments of the predicate were hashed into an integer value using the following function:

$$F = f(\text{pre } a_0 \ a_1) = |f(\text{pre})| + a_0 N + a_1 N^2$$

```
#define predicate_on 1
#define predicate_on_table 2
#define predicate_clear 3
#define predicate_hold 4
#define predicate_empty 5
#define total_predicates 5
```

We note that we have used absolute value for the predicate's corresponding value, this is because both `(on 1 2)` and `(~on 1 2)` have the same hash value which eases in finding if a state contains some proposition which might be negated in `preconditions` and `effects` of actions. Negative predicates do not occur in any state (which is represented using PDDL). This function resembles the base-N number and thus N must be greater than maximum f value (here, 5) and maximum block number($\max a_i$), thus $N = 1+\max(\text{total\_blocks}, \text{total\_predicates})$. We can then retrieve back the arguments in this way (`div` is integer division):

$$|f(\text{pre})| = F \bmod N$$

$$a_0 = (F \operatorname{div} N) \bmod N$$

$$a_1 = F \operatorname{div} N^2$$

```
typedef int proposition;

#define N (max(total_blocks, total_predicates) + 1)

#define type(x) (x % N)

#define var1(x) ((x / N) % N)
#define var2(x) (x / (N * N))
```

1.2.2 State

Now each state can be represented as a `set` of `proposition`s or equivalently integer hash values of them. For checking if a state is a goal state we can check for each `proposition` from goal state that whether it exitsts in the current state or not and becasue we have used `set` to represent states, thus this process becomes $O(\log n)$ (instead of $O(1)$, C++ uses Red-Black Tree).

```
typedef set<proposition> state;
```

1.2.3 Variable Action and Action

We can represent the action which contains variables in form of `variable_action` which has the following structure:

```
enum action_type {
        action_pick,
        action_unstack,
        action_release,
        action_stack
};

struct variable_action {
```

```
                action_type type;
                string name;
                vector<char> args;
                vector<condition> preconditions;
                vector<condition> effects;
};
```

The type and name are obvious, the args are actually variables starting from 'a' that represents the variables required by the `preconditions` and `effects` of the action, both of which cotain predicates in form of `condition`:

```
typedef int predicate;

struct condition {
                predicate _predicate;
                vector<char> args;

                inline proposition value(int var1, int var2,
                    int total_blocks) {
                        int k = N;
                        int v = abs(_predicate);
                        for (char c : args) {
                                v += k * ((c == 'a') ? var1 :
                                    var2);
                                k *= N;
                        }
                        return v;
                }
};
```

Here `predicate` is the corresponding value assigned to that predicate symbol above. `args` is similar to as above and `value` function returns the f value we previously discussed.

An action that has been instantiated can be assigned values to the arguments and thus becomes:

```
struct action {
                string name;
                vector<int> args;
                bool operator<(const action& that) const {
                        if (name.compare(that.name) >= 0)
                            return false;
                        if (args.size() >= that.args.size())
                            return false;
                        for (int i = 0; i < args.size(); i++)
                                if (args[i] >= that.args[i])
                                    return false;
```

```
                    return true;
                }
};
```

We have also defined `operator<` for comparison such that it can stored in a `set` (C++ uses Red-Black Tree for representation of sets internally.)

2 Forward Search using BFS