Chronological and Dependency-directed Backtracking

Mirna Udovičić

Department of Mathematics, University of Belgrade

Abstract: There are two ways to work on the faulty plan: chronologi-cal and dependency-directed backtracking. Chronological backtracking: you withdraw the most recen-tly made choice, and its consequences;find an alternative at that choice point, and move ahead again. Nonchronological backtracking: you withdraw the choices that matter.
There are many ways for doing search. One of the blind procedures is depth-first search. Depth-first search is a tree search: all possible paths are arranged in a search tree. Depth-first search: Given that one path is as good as any other, one way to find a path is to pick one of the children at every node visited, and to work forward from that child. Other alternatives at the same level are ignored completely, as long as it is possible to reach the goal. Depth-first search is efficient when unproductive partial paths are not too long.

1 Chronological Backtracking

Part of the depth-first procedure that responds to dead ends is called chronological backtarcking. Chronological backtracking is one way to work on a faulty plan; it begins as soon as a dead end is detected. The procedure is to withdraw the most recently made choice, and its consequences, to select an alternative at that choice point, and to move ahead again. If all the alternatives at the last choice point have been explored already, then go further back until an unexplored alternative is found.

The procedure is the following:

- Whenever you reach a dead end,
  - Until you encounter a choice point with an unexplored alternative,
    - Withdraw the most recently made choice
    - Undo all consequences of the withdrawn choice
    - Move forward again, making a new choice

The problem with chronological backtracking is clear: many of the withdrawn choices may have nothing to do with why the dead end is a dead end. Thus, chronological backtracking can be inefficient. In real problems, it can be impractical.

2 Nonchronological Backtracking

Another way to work on the faulty plan is to withdraw the choices that matter (the choices on which the dead end depends).

The procedure for identifying relevant choices is called dependency-directed backtracking, or nonchronological backtracking.

The procedure is:

- Whenever you reach an impasse,
- Trace back through dependencies,identifying all choice points that may have contributed to the impasse.
- Using depth-first search, find a combination of choices at those choice points that break the impasse.

Thus, nonchronological backtracking is an efficient way to find compatible choices, as long as there is a way of tracing back over dependencies to find the relevant choice points.

3 One Example of Nonchronological Backtracking: Weekly Schedule

Suppose we are given the next problem:

Each day of the week involves a set of choices for:

1 entertainment

2 exercise

3 study

- Tuesday, Wednesday and Thursday are study days.
- Monday and Friday are exercise days, and also entertainment days.

Monday	Tuesday	Wednesday	Thursday	Friday
exercise	study	study	study	exercise
entertainment				entertainment

Exercise choices are:

exercise	exercise units	expenses
walking	5	$0
jogging	10	$0
working out at a club	15	$20

Entertainment choices are:

entertainment	entertainment units	expenses
going to the restaurant	2	$20
reading a book	1	$0
doing nothing	0	$0

Study choices are:

study	study units
studying 0 hours	0
studying 2 hours	2
studying 4 hours	4
studying 6 hours	6

The problem is how to make a weekly schedule, having at least 6 hours of study, 2 units of entertainment, and 20 units of exercise. Expenses must be limited to $30 per week.

We will solve the problem using dependency-directed backtracking.

The first schedule could be any schedule, we have choosen this one:

Monday	Tuesday	Wednesday	Thursday	Friday
ex:walking	study 0 h	study 0 h.	study 0 h.	ex:walking
en:go to the r				en:go to the r

This plan is faulty, because, for example, expenses are $40, but they mustn't be bigger than $30. We must fix this plan, changing the choice which is connected with the problem. In this plan, that choice could be:

1 entertainment: going to the restaurant on Monday

2 entertainment: going to the restaurant on Friday.

We will change one entertainment choice to reading a book, or to doing nothing. After this change, the expenses are smaller, and it should check weather the new plan is the solution.

The study days: Tuesday, Wednesday, and Thursday are independent from the days for exercise and entertainment: Monday and Friday.

Thus, in the algorithm, there is a part Algorithm1 which gives a solution for Tuesday, Wednesday and Thursday.

The algorithm is given:

```
Algorithm
        begin
           Algorithm1;
           mon.ex = walking;
           mon.en = going_to_the_rest;
           fri.ex = walking;
           fri.en = going_to_the_rest;
           change(mon,fri);
        end.
    //Algorithm1 solves the problem of study units
    Algorithm1
        begin
            thu.st = 0;
            wed.st = 0;
            thur.st = 0;
            p = 0;
            while (not p)
              begin
                  thu.st = thu.st + 2;
                  p = (thu.st +wed.st +thur.st == 6);
                  if p break;
                  wed.st = wed.st + 2;
                  p = (thu.st +wed.st +thur.st == 6);
                  if p break;
                  thur.st = thur.st + 2;
                  p = (thu.st +wed.st +thur.st == 6);
              end;
        end;
    int checking (mon,fri);
         begin
             if ((expenses ≤ 30) and (exerciseunits ≥ 20)
                       and (entertunits ≥ 2))
                    return 1;
             else  return 0;
```

```
    end;
// function which returns 1 if there is a solution, and prints the result
int change(mon,fri)
    begin
       if (expenses > 30)
         begin
            if (fri.ex == working_at_a_club)
              begin
                 fri.ex = jogging;
                 p = checking(mon,fri);
                 if  p  begin    print(mon,fri);
                                 return 1;
                      end;
                 else  begin   p = change(mon,fri);
                               if  p  return 1;
                      end;
              end;
            if (fri.en == going_to_the_rest)
              begin
                 fri.en = reading_a_book;
                 p = checking(mon,fri);
                 if  p  begin    print(mon,fri);
                                 return 1;
                      end;
                 else  begin   p = change(mon,fri);
                               if  p  return 1;
                      end;
              end;
            if (mon.ex == working_at_a_club)
              begin
                 mon.ex = jogging;
                 p = checking(mon,fri);
                 if  p  begin    print(mon,fri);
                                 return 1;
                      end;
                 else  begin   p = change(mon,fri);
                               if  p  return 1;
                      end;
              end;
            if (mon.en == going_to_the_rest)
              begin
                 mon.en = reading_a_book;
                 p = checking(mon,fri);
                 if  p  begin    print(mon,fri);
```

```
                              return 1;
                         end;
                  else  begin   p = change(mon,fri);
                              if  p  return 1;
                         end;
              end;
       end;
  //if expenses are allowed, we are checking other conditions
```

4 Implementation

The algorithm is implemented in the programming language C++.

```
   //Enumeration type is used in this program(for weekly      //activities)
  enum vezbanje {setnja,trcanje,vezbaukl}
  enum zabava {veceranap,citknj,nista}
  //Class dani1 is used for Tuesday,Wednesday and //Thursday
  //Class dani2 is used for Monday and Friday
  class dani1 {
                 int u;
               }
 class dani2  {
                 vezbanje v;
                 zabava z;
                 int zadov;
                 int jedinvezb;
                 int ut;
               }

 int provera(dani2 pon, dani2 pet, int vezba, int zadov,int uktroskovi)
    { int p;
      p=((pon.jedinvezb+pet.jedinvezb >= vezba)&&
          (pon.zadov + pet.zadov >= zadov)&&
          (pon.ut+pet.ut <= uktroskovi));
      return p;
    }

 int zamena(dani2 pon, dani2 pet, int vezba, int zadov, int uktroskovi)
    { int q;
      int p;
      if (pon.ut + pet.ut > uktroskovi)
        {
         if (pet.v == vezbaukl)
```

```
          { pet.v = trcanje;
            pet.jedinvezb = 10;
            pet.ut = pet.ut – 20;
            q = provera(pon, pet, vezba, zadov, uktroskovi);
            if (q == 1) {
                        cout << "ponedeljak" << pon.v <<
                        " " << pon.z << endl;
                        cout << "petak" << pet.v << " " <<
                        pet.z << endl;
                          return 1;
                        }
    if  (!q)  { p = zamena( pon,pet, vezba, zadov, uktroskovi);
               if (p ==1) return 1;
              }
     }
        if (pet.z == veceranap )
           {  pet.z = citknj;
              pet.zadov = 1;
              pet.ut = pet.ut – 20;
                  ……
    //The rest of the function is similar to the beginning part;
    //the main difference is that we are checking other weekly
    //choices, changing them into other options (if that choices
    //cause a dead end)
```

Input: study units: 6, entertainment units: 2, exercise units: 20, expenses: 30$
Output: weekly schedule is:
Monday: exercise = jogging
entertainment = going_to_the_rest
Tuesday: studying 2 hours
Wednesday: studying 2 hours
Thursday: studying 2 hours
Friday: exercise = jogging
entertainment = reading_a_book .

Literature

[1] Artificial Intelligence, Patrick Henry Winston