2023年春形式化方法大作业

李远航 PB20000137

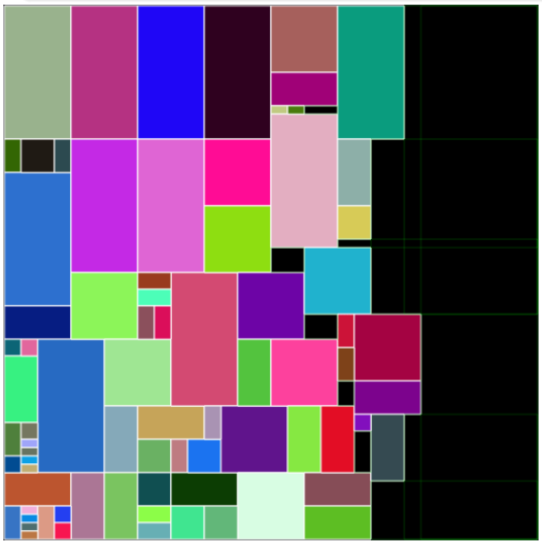
Rectangle-fitting

一、实验内容

分别用 Z3 和自己设计的算法求解 rectangle fitting

问题: Rectangle fitting

Given *a big rectangle* and *a number of small rectangles*, can you fit the small rectangles in the big one such that *no two overlap*.



How to specify this problem?

- Number rectangles from 1 to n
- for $i = 1 \ldots n$ introduce variables:
  - w_i is the width of rectangle i
  - h_i is the height of rectangle i
  - x_i is the x-coordinate of the left lower corner of rectangle i
  - y_i is the y-coordinate of the left lower corner of rectangle i

二、实验思路

- 使用 `z3` 求解（使用PPT中的思路）
  - 写出如下的约束表达式：

The following formula is *satisfiable* iff the fitting problem *has a solution*

$$\bigwedge_{i=1}^{n}((w_i = W_i \wedge h_i = H_i) \vee (w_i = H_i \wedge h_i = W_i))$$

$$\wedge \bigwedge_{i=1}^{n}(x_i \geq 0 \wedge x_i + w_i \leq W \wedge y_i \geq 0 \wedge y_i + h_i \leq H)$$

$$\wedge \bigwedge_{1 \leq i < j \leq n}(x_j \geq x_i + w_i \vee x_i \geq x_j + w_j \vee y_j \geq y_i + h_i \vee y_i \geq y_j + h_j)$$

  - 将上述约束转化成代码用 `z3` 求解：

```
# 约束 长方形的长宽,可以颠倒
for i in range(length):
    s.add(Or(And(W[i] == wlist[i], H[i] == hlist[i]),
          And(W[i] == hlist[i], H[i] == wlist[i]
# 约束 长方形的范围不超过最大
for i in range(length):
    s.add(X[i] >= 0)
    s.add(Y[i] >= 0)
    s.add(X[i] + W[i] <= maxw)
    s.add(Y[i] + H[i] <= ma
# 约束 长方形不覆盖
for i in range(length):
    for j in range(i+1, length):
        s.add(Or(X[j] >= X[i]+W[i], X[i] >= X[j]+W[j],
              Y[j] >= Y[i]+H[i], Y[i] >= Y[j]+H[j]))
```

- 使用自己的方法求解
  - 由于Rectangle-fitting实际上是一个NP-hard问题， 所以正常方法难以求解，这里使用搜索策略，通过递归回溯的方式求解
  - 每一个小长方形的左下角一定在格点上，由于小长方形的安排顺序并不影响最终的结果，所以依次递归遍历每一个格点，一定可以找到如果存在的解
  - 为了避免不必要的遍历，在程序开始前，将所有长方形的长宽同时除以最大公因子
  - 优先安排面积最大的小长方形
  - csp约束条件：

```
def isvalid(wlist, hlist, x, y):
    l = len(x)
    for i in range(l):
        for j in range(i+1, l):
            if x[j] < x[i]+wlist[i] and x[i] < x[j]+wlist[j] and
            y[j] < y[i]+hlist[i] and y[i] < y[j]+hlist[j]:
                return False
    return True
```

  - 递归函数：

```
def solve_recursion(maxw, maxh, wlist, hlist, now_x, now_y):
    if not isvalid(wlist, hlist, now_x, now_y):
        return False, [], []

    now_loc = len(now_x)
    if now_loc == len(wlist):
        print(wlist, hlist)
        return True, now_x, now_y

    now_w, now_h = wlist[now_loc], hlist[now_loc]
    for i in range(maxw):
        for j in range(maxh):
            if i + now_w <= maxw and j + now_h <= maxh:
                now_x.append(i)
```

```
                now_y.append(j)
                ret = solve_recursion(maxw, maxh, wlist, hlist, now_x,
now_y)
                if ret[0]:
                    return True, ret[1], ret[2]
                now_x = now_x[:now_loc]
                now_y = now_y[:now_loc]

    wlist[now_loc], hlist[now_loc] = hlist[now_loc], wlist[now_loc]
    now_w, now_h = wlist[now_loc], hlist[now_loc]
    for i in range(maxw):
        for j in range(maxh):
            if i + now_w <= maxw and j + now_h <= maxh:
                now_x.append(i)
                now_y.append(j)
                ret = solve_recursion(maxw, maxh, wlist, hlist, now_x,
now_y)
                if ret[0]:
                    return True, ret[1], ret[2]
                now_x = now_x[:now_loc]
                now_y = now_y[:now_loc]

    return False, [], []
```

三、实验结果

使用测试数据：

```
7 7
2 1
2 1
2 1
2 1
1 1
5 2
5 2
5 2
5 2
```

可以看到，两种方法均能求出答案:

```
X0 = 6 Y0 = 3 W0 = 1 H0 = 2
X1 = 6 Y1 = 1 W1 = 1 H1 = 2
X2 = 0 Y2 = 6 W2 = 2 H2 = 1
X3 = 0 Y3 = 0 W3 = 2 H3 = 1
X4 = 6 Y4 = 0 W4 = 1 H4 = 1
X5 = 2 Y5 = 5 W5 = 5 H5 = 2
X6 = 2 Y6 = 0 W6 = 2 H6 = 5
X7 = 0 Y7 = 1 W7 = 2 H7 = 5
X8 = 4 Y8 = 0 W8 = 2 H8 = 5
[5, 5, 5, 2, 2, 2, 2, 2, 1] [2, 2, 2, 5, 1, 1, 1, 1, 1]
((True, [0, 0, 0, 5, 0, 2, 4, 5, 6], [0, 2, 4, 0, 6, 6, 6, 5, 6]), 1)
```

性能比较：（测试数据见 `input` 目录）

	input0	input1	input2	input3(无解)
z3	0.01844s	0.03959s	1.26748s	1.02537s
recursion	0.00008s	0.00040s	0.00404s	5.77381s

可以看到

- 由于搜索策略的使用，递归回溯的方法相比 `z3` 求解能有更高的性能
- 对于 `input3` 无解的情形，由于递归算法需要遍历整个取值空间，性能急剧下降，远低于 `z3` 求解
- 在解决现实问题时，为了应对多样化的需求，应该尽量使用 `z3` 求解，避免极端数据

四、实验收获

- 使用SMT解决问题思路清晰，且性能高效
- 使用搜索的方式解决约束满足问题在策略合理的情况下，一样能达到不错的效率

设计CNF的SAT求解算法

一、实验内容

设计CNF的SAT求解算法

- 可以使用现有算法 (如 DPLL, CDCL)，也可以自行设计其他算法
- 可以独立设计可执行程序，也可以修改现有开源程序的核心算法 (选取后者分数更高)
- 自己构建测试集（可网上查找测试集）
- 附上详细的文档: 包括实现过程，算法解释，与现有工具 (如 Z3) 等的性能对比

二、实验思路

使用DPLL算法实现，DPLL算法的伪代码如下：

```
Algorithm DPLL(CNF Φ) :=
    do UP(Φ) until It changed nothing.
    do PLE(Φ) until It changed nothing.
    if Φ = ∅ then
        return true.
    if ∃L ∈ Φ, L = ∅ then
        return false.
    x ← ChooseVariable(Φ)
    return DPLL(Φ_{x→true}) or DPLL(Φ_{x→false})
```

对整个CNF表达式使用单位字句传播，例如：

$$(a \vee b \vee c \vee \neg d) \wedge (\neg a \vee c) \wedge (\neg c \vee d) \wedge (a)$$

表达式中 a 单位字句，显然原表达式为真需要 a 变量为真，再由于 a 的值已经取定，可以借此化简原CNF表达式成为：

$$(c) \wedge (\neg c \vee d)$$

而对于无法直接判断的变量，使用深度优先搜索，完成对值的查找过程，具体实现的 `python` 代码如下：

```python
def dpll(var, cnflist, orderlist):
    if len(cnflist) == 0 or len(orderlist) == var:
        if len(cnflist) == 0:
            retlist = []
            for i in orderlist:
                retlist.append(i if orderlist[i] else -i)
            return retlist
        else:
            return []
    jdg_this = -1
    for cnf_text in cnflist:
        if len(cnf_text) != 0:
            jdg_this = abs(cnf_text[0])
        if len(cnf_text) == 1:
            orderlist[abs(cnf_text[0])] = (cnf_text[0] > 0)
            break

    basecnf = copy.deepcopy(cnflist)

    if jdg_this in orderlist:
        for cnf_text in basecnf:
            if jdg_this in cnf_text:
                if orderlist[jdg_this] == True:
                    cnflist.remove(cnf_text)
```

```
                else:
                    cnflist[cnflist.index(cnf_text)].remove(jdg_this)
            elif -jdg_this in cnf_text:
                if orderlist[jdg_this] == False:
                    cnflist.remove(cnf_text)
                else:
                    cnflist[cnflist.index(cnf_text)].remove(-jdg_this)
        return dpll(var, cnflist, copy.deepcopy(orderlist))
    else:
        orderlist[jdg_this] = True
        for cnf_text in basecnf:
            if jdg_this in cnf_text:
                cnflist.remove(cnf_text)
            elif -jdg_this in cnf_text:
                cnflist[cnflist.index(cnf_text)].remove(-jdg_this)
        ret = dpll(var, cnflist, copy.deepcopy(orderlist))
        if ret:
            return ret
        cnflist = copy.deepcopy(basecnf)
        orderlist[jdg_this] = False
        for cnf_text in basecnf:
            if jdg_this in cnf_text:
                cnflist[cnflist.index(cnf_text)].remove(jdg_this)
            elif -jdg_this in cnf_text:
                cnflist.remove(cnf_text)
        return dpll(var, cnflist, copy.deepcopy(orderlist))
```

三、实验结果

本程序使用标准的CNF输入格式，即第一行表明CNF表达式的变量数，子句数，后续输入表达式，如下所示，表示输入有三个变量，6个子句

```
p cnf 3 6
1 2 0
1 -2 0
3 2 0
-3 1 0
1 2 3 0
-1 -2 0
```

这里使用 `python-sat` 验证代码运行结果的准确性，如下所示

```
py SAT    [1, -2, 3]
my dpll   [1, -2, 3]
```

可以看到，程序运行的结果相同，验证了程序的正确性

下面比较程序运行的效率，这里选用了两组测试数据，一组有解，一组无解，分别运行5000次，观察程序的运行时间：

	input0（有解）	input1（无解）
python-sat	0.0825789s	0.0915676
my-dpll	0.1068725	0.9983847

可以看到，自主实现的求解算法程序性能弱于`python-sat`，其原因可能是在不必要的搜索分支上浪费了大量的时间，以及`python-sat`可能使用了更优秀的剪枝策略

四、实验收获

- 对DPLL算法有了更深的理解
- 亲自实现的DPLL算法求解CNF的性能远远不如现代化工具，还有更多需要学习的地方