Report

Ye Zhentao 2101111526

2022 年 2 月 3 日

1 DONE

a. 对于 benchmark 上的部分例子（某个.sl 文件），根据最后得到的解的形式，添加可以一步求解的 compose 组件和删除其他一些用不到的组件，观察 maxflash 在这样的新 DSL 上的表现。

b. 基于 krasserm 博客提供的 GP+BAY 的框架，从一维连续的应用扩展到多维连续再扩展到多维二值，即定义域为 $\{0,1\}^n$，每一位表示某一个修改操作是否应用。

2 TODO

c. 抽象出各种修改操作的表达形式，并写一个 constructor 用于接收原 DSL 和某一个修改操作，给出训练集上将语言替换为修改后的 DSL 的所有.sl 文件。

d. 求解器和采样搜索框架的接口

e. 进一步修改 BAY 的框架

3 遇到的问题

3.1

为什么没有先做单纯工程性的 c？因为对 a 的观察发现了一些不符合预期的问题。具体来说看以下例子：

```
(define-fun doublerep ((x String) (s String) (t String)) String (str.replace (str.replace x s t) s t))
(synth-fun f ((name String)) String
    (
     (ntString String (name " " "(" ")" "-" "."
                       (str.++ ntString ntString)
                       (doublerep ntString ntString ntString)
                       (str.at ntString ntInt)
                       (int.to.str ntInt)
                       (str.substr ntString ntInt ntInt)))
```

图 1: doublerep

这个例子相比于源语言增加了 doublerep 并且删除了原有的 str.replce。源语言上的解正是 $f : str.replace(str.replace(Param0, " - ", "."), " - ", ".")$,对应到新语言上即为 $f : doublerep(Param0, " - ", ".")$。

理论上我们希望通过这样的组件缩短搜索路径，或者再加上删除一些其他用不到的组件进一步提高效率。

但是直接用 maxflash 对这两个样例求解的结果显示：源语言求解时间稳定在 0.0052s，新语言求解时间稳定在 0.012s。和吉老师讨论过一会但是我并没有很理解为什么：如果这样的基本的组件修改不能提高效率，我们应该期待的可以提高效率的结果是怎样的？我觉得至少要手动把所需要的组件修改得到新 DSL 相比于原 DSL 效果要好的实验结果观测到，才能寄希望于采样搜索算法可以能帮我们找到这个解。

在其他的样例上也遇到了类似的情况，如初始解为
$f : str.++(Param0, str.++("/n", str.++(str.++(Param1, "/n"), Param2)))$ 的 addalinebreak.sl，添加进 str.++ 的重复组合后求解时间大大增加。（从 0.2s 增加到了 50s）

这里还有另一个问题需要考虑：如上述要把四个 str.++ 组合成一个组件时，实际上这个 str4plus 需要接收的是 5 个参数，那么这样参数搜索空间的增大是不是导致了效率大大降低？前面 doublerep 利用了最后解的特点，即两次接受的参数相同，而这个信息依赖于频繁模式挖掘。换句话说，现在如果要手动暴力定义出所有可能的组件，根据参数的相同或不同写多个同样结构的组件（相当于对参数也加了一个限制条件比如 para1=para3）可能才能对结果有提升？

minus：maxflash 求解是不是还对基本组件的最小个数有要求，当我把所有其他产生式都删除只留下 doublerep 或者 str4plus 的时候会直接报段错误。

总结：需要怎么手动得到一个源 DSL 在某一个例子上效率不如进行了某种修改后的新 DSL 的例子，才能尝试自动化这样的搜索。

3.2

这里主要是在做 b 描述的扩展中遇到的问题。首先，我们现在想要解决的问题可以抽象为：有一个函数 f 定义在 n 维单位立方体的端点上，且我们大概知道这个函数满足，互相接近的端点对应的函数值相对的也接近（两个端点之间的接近被描述为从一个点走到另一个点的步数），最终目标是通过较少的采样次数，推测得到函数 f 的极值点或者相对让函数值接近极值的点。

而标准的通过高斯过程 + 贝叶斯优化的框架流程如下：
1. 目标函数 f，通过初始采样 $D = \{x_i, y_i\}$ 建立高斯模型
2. 通过对采集函数（acquisition function）求极值指导下一个采样点，经典的比如 EI 采集函数就描述了在当前高斯模型基础上，在某一个点 x 采样所能得到的预期改进，这个改进可以分割成：在已有最大值附近得到更大值的可能性和在未知性较高的区域采样增强对函数 f 的整体刻画。
3. 将新数据点加入 D 并重复此过程。

理论上我们的问题的确也是一个符合它应用场景的黑盒优化问题。在实现的修改上遇见的问题如下：（代码见图 2code）

这一部分代码想要解决的问题是，已经有了一个 EI 函数描述了任意一个采样点 X 对当前模型的贡献大小，想要找到贡献最大的一个点当做下一个采样点。

114 行-123 行是原本对于一维或者多维的连续定义域（EI 的定义域与高斯过程描述的定义域相同，如在我们的问题里也是 $\{0,1\}^n$）求 EI 极值的方法，直接利用了 scipy.optimize 中的 minimize 算法，我粗略的学习一下后感觉对于我们每一维只有两个取值的情况并不适用。所以这里需要解决的问题是，在 $\{0,1\}^n$ 上快速找到函数极值点的方法（我觉得应该有人做过?），图里的代码先做了个用限定步数的重新设置初值加随机搜索 n_search 步的方法代替。

总结：在 $\{0,1\}^n$ 上快速找到函数极值点的方法是什么？

minus：其实也还有几个其他不同的采集函数各自有不同的理论保证，如 UCB，POI，但这些

```
    def min_obj(X):
        # Minimization objective is the negative acquisition function
        return -acquisition(X.reshape(-1, dim), X_sample, Y_sample, gpr)

    # Find the best optimum by starting from n_restart different random points.
    # for x0 in np.random.uniform(bounds[:, 0], bounds[:, 1], size=(n_restarts, dim)):
    #     res = minimize(min_obj, x0=x0, bounds=bounds, method='L-BFGS-B')
    #     if res.fun < min_val:
    #         min_val = res.fun[0]
    #         min_x = res.x

    for i in range(n_restarts):
        rd = np.random.rand(n)
        x0 = rd.round().astype(int)
        res = min_obj(x0)
        for j in range(n_search):
            step_to = np.random.randint(n)
            x1 = x0.copy()
            x1[step_to] = 1 - x1[step_to]
            res1 = min_obj(x1)
            if res1 < res:
                res = res1
                x0 = x1.copy()
        if res < min_val:
            min_val = res
            min_x = x0
```

图 2: code

采集函数最后都逃不开一个求它的极值去指导下一个采样点的问题。

4 总结

TODO 中的 c 和 d 为独立工程问题，d 比较简单，但是 c 我认为应该先把 3.1 的问题解决，理解了怎样的修改操作可以改进求解速度后才可以定义出包括目标修改操作的一大堆修改，然后再扔给搜索过程。（师兄原论文里提出的提高效率的例子是将 str.replace 添加进了本来没有这个函数的语言里，虽然也提到了 replace 可以由原来的组件表达出来，但是实际上调用 replace 的运行逻辑是不是并不是直接跑那一大堆原组件?）

TODO 的 e 可以在解决了 3.2 的问题后（有更好的求 EI 极值的方法）做改进。