AI+ 软件研发数字峰会
AI+ software Development Digital summit

AI驱动软件研发
全面进入数字化时代

中国·北京 08.18-19

AI+
software
Development
Digital
summit

算法合成——
自动应用算法模式合成高效程序

熊英飞 北京大学

科技生态圈峰会 + 深度研习 ——1000 + 技术团队的共同选择

K+峰会

2023K+
全球软件研发行业创新峰会
上海站
会议时间 | 06.09-10

2023K+
全球软件研发行业创新峰会
北京站
会议时间 | 07.21-22

2024K+
全球软件研发行业创新峰会
深圳站
会议时间 | 05.17-18

K+峰会详情

AiDD峰会

会议时间 | 08.18-19
AiDD AI+软件研发数字峰会
北京站

会议时间 | 11.17-18
AiDD AI+软件研发数字峰会
深圳站

AiDD峰会详情

演讲嘉宾

熊英飞

北京大学副教授

熊英飞于2009年从日本东京大学获得博士学位，2009-2011年在加拿大滑铁卢大学工作，2012年加入北京大学，现任新体制长聘副教授。熊英飞的研究兴趣是程序设计语言和软件工程，特别是程序合成、修复和分析。他提出了理论和方法降低程序编写和缺陷修复的代价。比如，基于差别的双向变换框架是最广泛使用的双向变换框架之一，概率和逻辑结合的程序合成框架玲珑框架将程序修复的正确率从此前不到40%提升到80%以上。他的工作也被工业界采用，比如新一代Linux内核配置项目、燕云DaaS系统、华为公司等。他获得CCF-IEEE CS青年科学家奖、MODELS十年最有影响力论文奖，5次获得ACM SIGSOFT/IEEE TCSE杰出论文奖。他是SATE18的程序委员会联合主席，也在PLDI、ICSE、FSE等会议担任PC。

知乎问题

程序员 软件开发 编程 计算机 计算机经典课程

作为计算机专业学生，最应该学习的课程前五位是什么？

不论当前大学是否开设这些课程。

关注问题 写回答 邀请回答 好问题 371 13 条评论

1、数据结构与算法

2、计算机组成原理

3、操作系统

4、计算机网络

5、数据库

▲ 赞同 1461 ▼ 124 条评论

软件工程 计算机科学 计算机专业

你认为计算机专业最难学的课程是什么？

关注问题 写回答 邀请回答 好问题 16

课程名	点赞数
算法数据结构	47
大学语文	23
马克思主义	12
汇编语言	5
体系结构	1

算法为什么难：示例

- 最大子段和问题:
  - 输入一个整数的列表，
  - 求列表上连续一段的最大和。
  - [1, -2, 3, -2, 3] → 4

- 穷举算法非常容易实现
- 算法复杂度为$O(n^3)$
- 能否应用算法课的知识
  来进行优化?

```
mss = -INF
for i in range(len(xs)):
  for j in range(i, len(xs)):
    mss = max(mss, sum(xs[i: j + 1]))
return mss
```

算法为什么难：示例

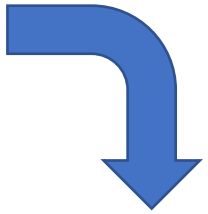
- 算法课教了一系列算法设计模式，尝试应用分治
- 算法复杂度: $O(n/m)$, m: 处理器个数
- 应用算法设计模式并不容易
  - 分治：把原问题分解成规模更小的问题
    - 对具体问题并没有给出分解方法，需要程序员的智慧
  - 分治后的程序长度、理解难度均远超原来的程序

```
def aux(x):
  mps = max([sum(x[:i+1]) for i in range(lens(x))])
  mts = max([sum(x[i:]) for i in range(lens(x))])
  sumx = sum(x)
  return (mps, mts, sumx)
def comb(L, R):
  (mssL, (mpsL, mtsL, sumL)) = L
  (mssR, (mpsr, mtsR, sumR)) = R
  mss = max(mssL, mssR, mtsL + mpsR)
  mps = max(mpsL, sumL + mpsR)
  mts = max(mtsL + sumR, mtsR)
  sumx = sumL + sumR
  return (mss, (mps, mts, sumx))
```

```
def dac(x, l, r):
  if l + 1 <= r:
    return (orig([x[l]]), aux([x[l]]))
  mid = (l + r) // 2
  parallel:
    lres = dac(x, l, mid)
    rres = dac(x, mid, r)
  return comb(lres, rres)
return dac(x, 0, len(x))[0]
```

研究目标：自动应用算法模式

```
mss = -INF
for i in range(len(xs)):
  for j in range(i, len(xs)):
    mss = max(mss, sum(xs[i: j + 1]))
return mss
```



应用分治

```
def dac(xs, l, r):
  if l + 1 <= r
    return (xs[l], (xs[l], xs[l], xs[l]))
  mid = (l + r) // 2
  parallel:
    resL = dac(x, l, mid)
    resR = dac(x, mid, r)
  return comb(resL, resR)
return dac(xs, 0, len)[0]
```

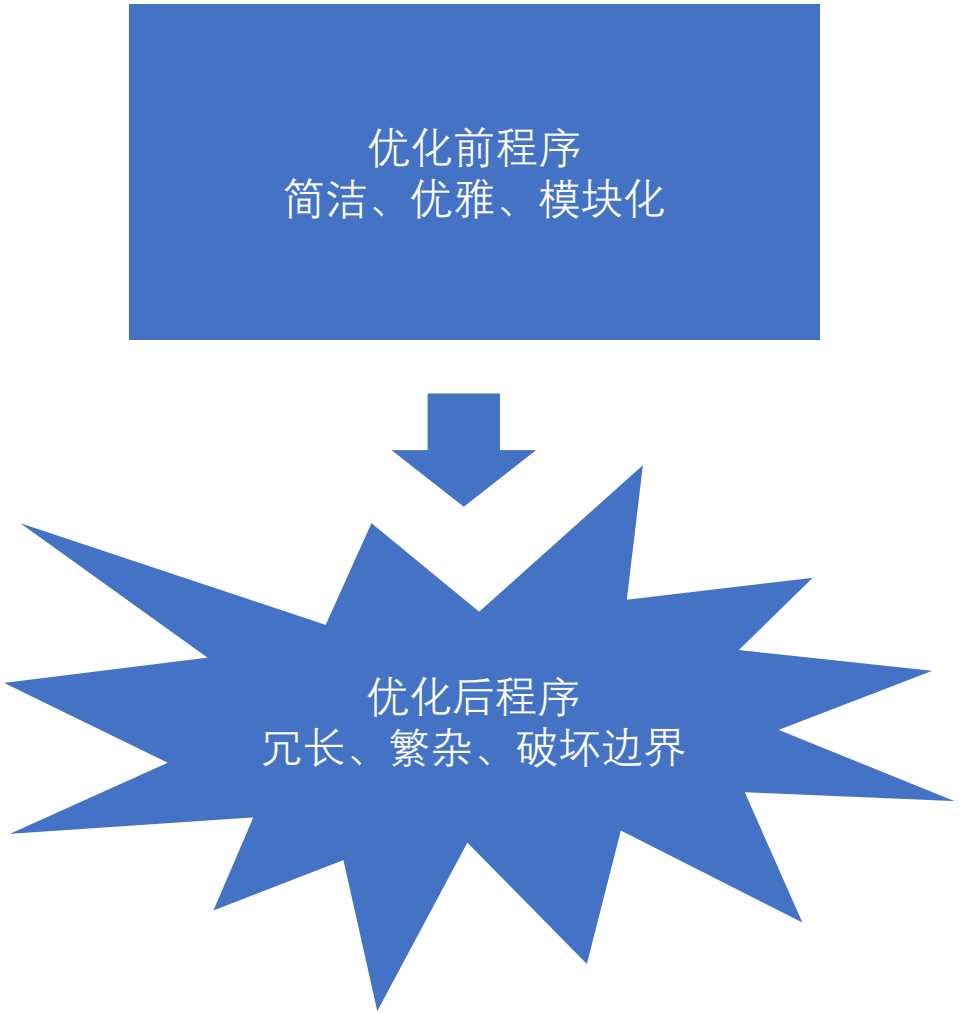
```
def comb(resL, resR):
  (mssL, (mpsL, mtsL, sumL)) = resL
  (mssR, (mpsR, mtsR, sumR)) = resR
  mss = max(mssL, mssR, mtsL + mpsR)
  mps = max(mpsL, sumL + mpsR)
  mts = max(mtsL + sumR, mtsR)
  sumx = sumL + sumR
  return (mss, (mps, mts, sumx))
```

降低难度：算法设计是困难耗时的步骤

节省成本：会算法的程序员需要更高人力成本

同一时间、同一地点、同一公司、同一部门的两个招聘岗位

提高质量：算法优化是缺陷来源



- SeL4: 著名经验证操作系统内核
  - 第一阶段：采用Haskell的算法设计
    - 验证导致500+修改
  - 第二阶段：采用C的系统实现
    - 验证导致50+修改

提升效率：自动优化程序

目前的编译优化技术只能进行局部小替换，通常很难显著降低程序复杂度。自动应用算法模式有望对程序进行整体大幅优化。

数据流敏感重写 flow-sensitive rewrites

- 条件常量传播 conditional constant propagation
- 主导测试检测 dominating test detection
- 基于流承载的类型缩减转换 flow-carried type narrowing
- 无用代码消除 dead code elimination

语言相关的优化技术 language-specific techniques

- 类型继承关系优化 class hierarchy analysis
- 去虚拟机化 devirtualization
- 符号常量传播 symbolic constant propagation
- 自动装箱消除 autobox elimination
- 逃逸分析 escape analysis
- 锁消除 lock elision
- 锁膨胀 lock fusion
- 消除反射 de-reflection

内存及代码位置变换 memory and placement transformation

- 表达式提升 expression hoisting
- 表达式下沉 expression sinking
- 冗余存储消除 redundant store elimination
- 相邻存储合并 adjacent store fusion
- 卡痕消除 card-mark elimination
- 交汇点分离 merge-point splitting

循环变换 loop transformations

- 循环展开 loop unrolling
- 循环剥离 loop peeling
- 安全点消除 safepoint elimination
- 迭代范围分离 iteration range splitting
- 范围检查消除 range check elimination
- 循环向量化 loop vectorization

全局代码调整 global code shaping

- 内联 inlining (graph integration)
- 全局代码外提 global code motion
- 基于热度的代码布局 heat-based code layout
- switch 调整 switch balancing
- 抛出内联 throw inlining

常见编译优化技术

大型语言模型是否解决这个问题?

Can you optimize this program as a parallel program using D&C? The expected time complexity is O(n/p), where n is the length of the input list x, and p is the number of CPU cores.

```
mss = -INF
for i in range(len(x)):
  for j in range(i, len(x)):
    mss = min(mss, sum(x[i: j+1]))
return mss
```

min而不是max

```
import threading

# Define a global variable for infinity
INF = float('inf')

# Define a helper function that returns min(a,b)
def min(a,b):
  if a < b:
    return a
  else:
    return b

# Define a recursive function that returns
#the minimum sum subarray of x[low:high+1]
def mss(x, low, high):
  # Base case: only one element
  if low == high:
    return x[low]
```

```
# Find middle index
mid = (low + high) // 2

# Create two threads for left and right halves
t1 = threading.Thread(target=mss,args=(x, low,mid))
t2 = threading.Thread(target=mss,args=(x,mid+1 ,high))

# Start both threads
t1.start()
t2.start()

# Wait for both threads to finish
t1.join()
t2.join()

# Get their results
left_mss = t1.result
right_mss = t2.result
```

Learn more: 1. techiedelight.com 2. geeksforgeeks.org 3. link.springer.com

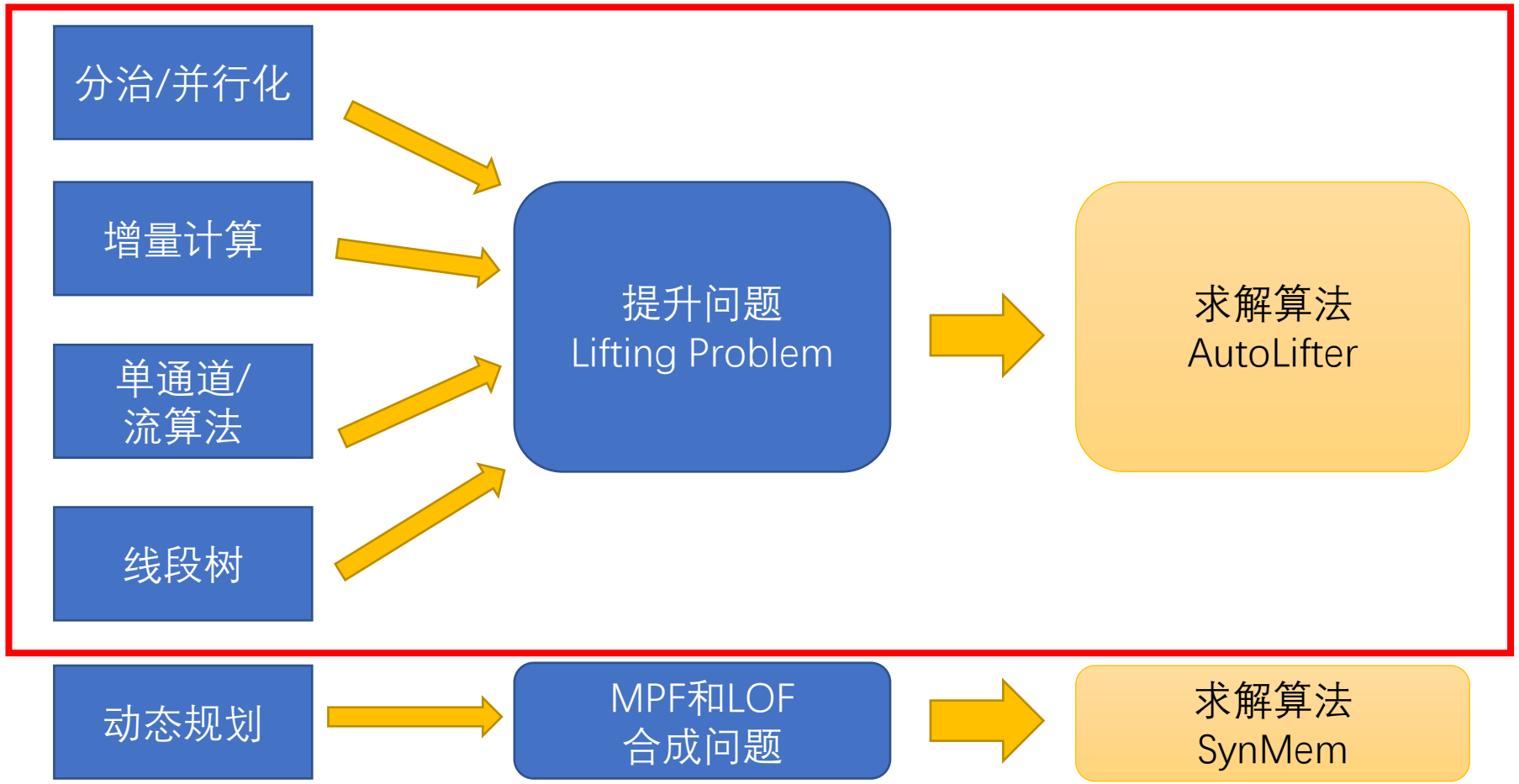
北京大学目前成果

- 两类算法设计模式的自动应用方法
  - 分治类问题[arXiv:2202.12193]
    - 分治（并行化）、增量计算、流算法、线段树、最长子段问题等
  - 动态规划问题[OOPSLA23]
  - 根据不同的算法设计模式实例化成不同的合成工具
- 例：列表上的$O(\frac{n}{m})$分治算法的合成工具
  - 输入：
    - 一个程序，从列表产生输出
      - 可以用任意编程语言、任意算法编写
      - 也可从形式化需求自动导出
  - 输出：
    - 一个列表上的分治程序
      - 和输入程序等价
      - 算法复杂度是$O(\frac{n}{m})$，其中m为CPU的数量

传统求解方法：程序演算

- 通过一系列程序变换半自动得到优化后的程序
- 但算法设计往往需要“根据具体问题设计关键模块”这样的操作
  - 通用程序变换规则难以完成问题特定的设计
- 因此，程序演算保持半自动求解

方法概览


分治/并行化
增量计算
单通道/
流算法
线段树
提升问题
Lifting Problem
求解算法
AutoLifter
动态规划
MPF和LOF
合成问题
求解算法
SynMem
规约
基于搜索的
归纳程序合成求解

手动应用分治算法

- 第二小问题：
  - 输入整数列表
  - 查找列表中的第二小的元素

- 输入程序

```
return sorted(xs)[1]
```

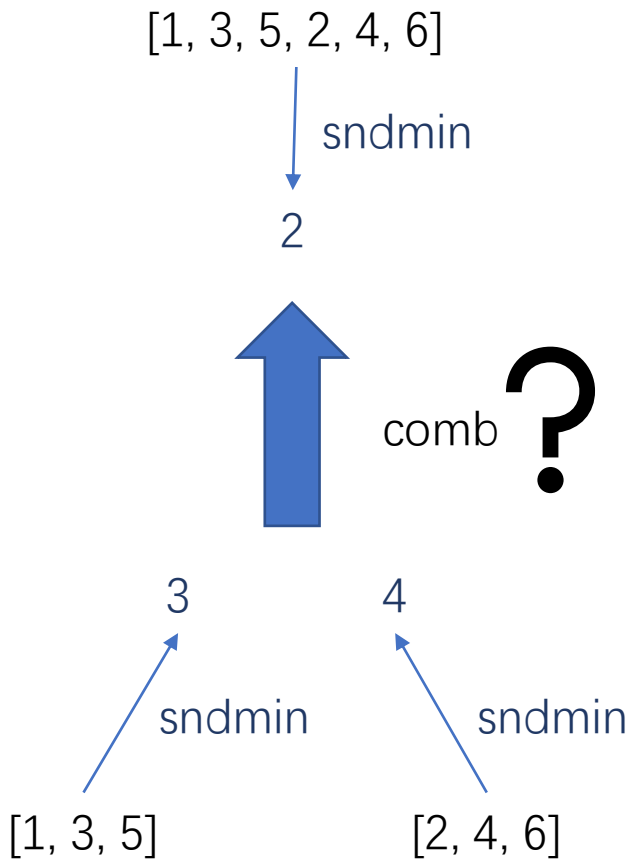
  - 时间复杂度: O(nlogn)

- 目标：采用分治优化程序
  - 得到O(n/m)并行程序

手动应用分治算法

- 分三步应用分治:
  1. 将输入列表xs分成两份$xs_L + xs_R$
  2. 在两份列表上分别递归计算*sndmin*
  3. 合并结果得到原列表xs上的结果

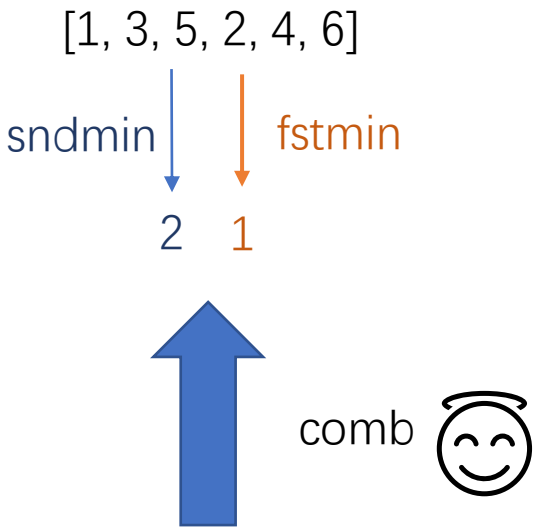
- 但是，这样的合并操作是不存在的



手动应用分治算法

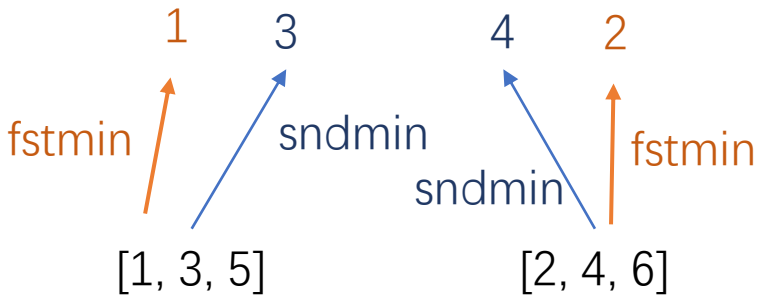
- 从原列表中产生辅助值

$$aux(xs) = \min(xs)$$



- 然后可以写出合成函数

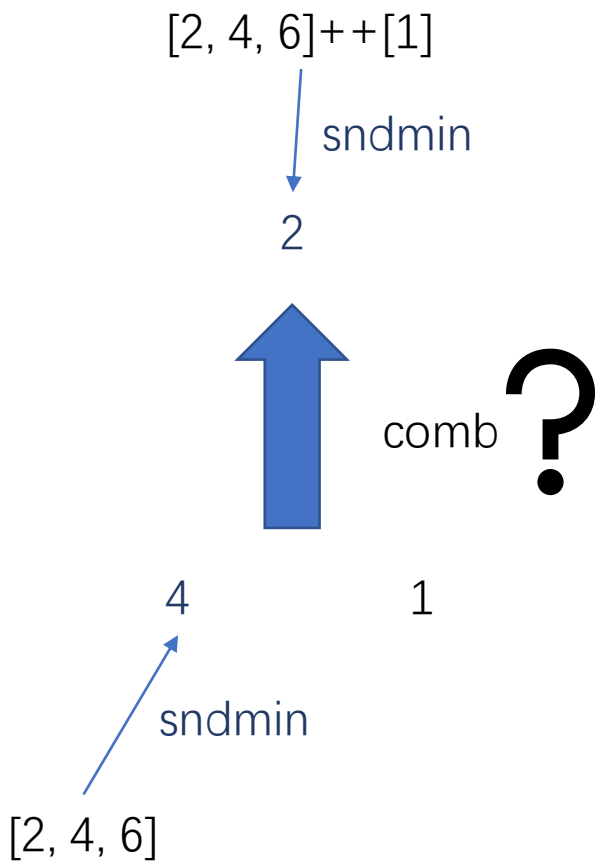
$$comb\left((snd_L, fst_L), (snd_R, fst_R)\right) = (\min(snd_L, snd_R, \max(fst_L, fst_R)), \min(fst_L, fst_R))$$



寻找一个辅助函数aux，使得可以找到comb函数，将两个子列表上算出来的结果合并为原列表的结果

手动应用增量算法

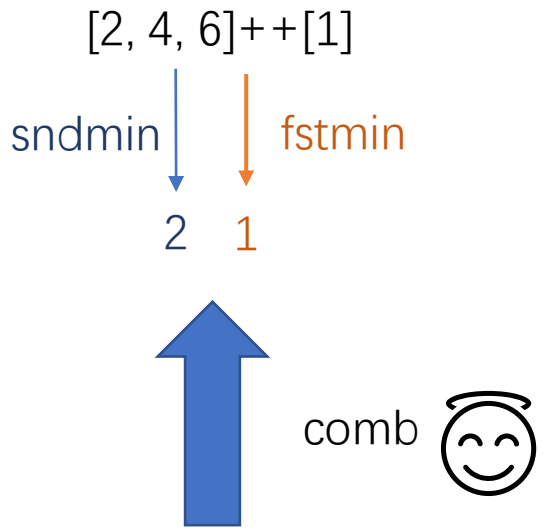
- 问题:
  - 假设已经对一个很长的列表求出了第二小
  - 现在列表后面新增了一个元素
  - 如何快速计算新列表的第二小?
- 和之前类似，这样的合并函数不存在
- 增量算法通常从原输入产生辅助值使得结果可以快速更新



手动应用增量算法

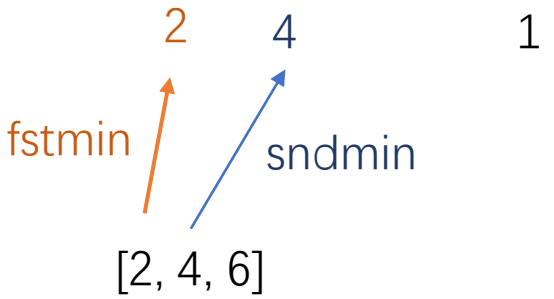
- 从原列表中产生辅助值

$$aux(xs) = \min(xs)$$



- 然后可以写出合成函数

$$comb((snd, fst), v)$$
$$= (\min(snd, \max(fst, v)), \min(fst, v))$$



寻找一个辅助函数aux，使得可以找到comb函数，将原列表上算出来的结果和修改合并为新列表的结果

提升问题Lifting Problem

- 输入:
  1. a: 某种数据结构的实例
  2. c: 附加值
  3. orig: 从数据结构上计算某种结果
  4. op: 从已有数据结构实例构造新实例
- 输出: 满足下面条件的辅助值程序 aux 和合并程序 $comb$
  - $orig'\big(op(c, a_1, \ldots, a_n)\big) = comb\big(c, orig'(a_1), \ldots, orig'(a_n)\big)$

$$where\ orig'(a) = (orig(a), aux(a))$$

$orig$: $sndmin$

列表上的分治

$c = ()$

$a_1 = [1, 3, 5]$

$a_2 = [2, 4, 6]$

op: $a_1 + a_2$

提升问题Lifting Problem

- 输入:
  1. a: 某种数据结构的实例
  2. c: 附加值
  3. orig: 从数据结构上计算某种结果
  4. op: 从已有数据结构实例构造新实例
- 输出: 满足下面条件的辅助值程序 aux 和合并程序 $comb$
  - $orig'\big(op(c, a_1, \dots, a_n)\big) = comb\big(c, orig'(a_1), \dots, orig'(a_n)\big)$

增量算法

$c = 1$

$a_1 = [2, 4, 6]$

$op: a_1 + [c]$

$where\ orig'(a) = (orig(a), aux(a))$

$orig: sndmin$

提升问题Lifting Problem

- 输入:
  1. a: 某种数据结构的实例
  2. c: 附加值
  3. orig: 从数据结构上计算某种结果
  4. op: 从已有数据结构实例构造新实例
- 输出: 满足下面条件的辅助值程序 aux 和合并程序 $comb$
  - $orig'(op(c, a_1, \dots, a_n)) = comb(c, orig'(a_1), \dots, orig'(a_n))$

$$where\ orig'(a) = (orig(a), aux(a))$$

更通用的
增量算法

$c =$ *"delete 2nd element"*

$a_1 = [2, 4, 6]$

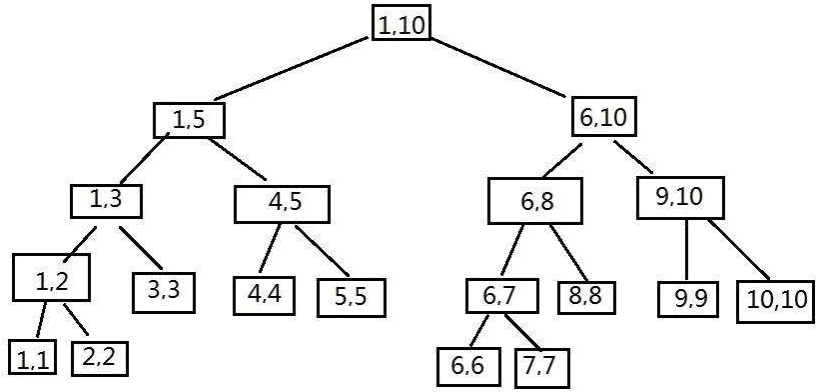
$op: apply(c, a_1)$

$orig: sndmin$

线段树

- 高效回答子段查询/修改操作的数据结构
  - 查询：序列中第4-5000个数的第二小是多少?
  - 修改：将序列第2-3000个数都加一

  - 通过分治来完成查询
  - 通过增量算法来完成修改



$$op\big((tag,c),(a_1,a_2)\big) := \begin{cases} a_1 + a_2 & tag = 1 \\ apply(c,a_1) & otherwise \end{cases}$$

提升问题Lifting Problem

- 其他很多算法模式的应用也可以转成提升问题
  - 分治/并行化
  - 增量计算
  - 流算法
  - 线段树
  - 最长子段问题
  - 动态规划（进行中）
  - ……

如何求解提升问题?
背景：程序合成

- 从规约中自动生成程序

“One of the most central problems in the theory of programming.”

----Amir Pneuli

图灵奖获得者

“（软件自动化）提升软件生产率的根本途径”

----徐家福先生

中国软件先驱

程序合成问题定义

- 输入:
  - 一个程序空间*Prog*, 通常用语法表示
  - 一条规约*Spec*, 通常为逻辑表达式
- 输出:
  - 一个程序*prog*，满足
    - $prog \in Prog \wedge prog \mapsto Spec$

例子：max问题

- 语法：

$$
\begin{array}{rcl}
\texttt{Expr} & ::= & 0 \mid 1 \mid x \mid y \\
& \mid & \texttt{Expr} + \texttt{Expr} \\
& \mid & \texttt{Expr} - \texttt{Expr} \\
& \mid & (\texttt{ite BoolExpr Expr Expr}) \\
\texttt{BoolExpr} & ::= & \texttt{BoolExpr} \wedge \texttt{BoolExpr} \\
& \mid & \neg\texttt{BoolExpr} \\
& \mid & \texttt{Expr} \leq \texttt{Expr}
\end{array}
$$

- 规约：

$$
\begin{aligned}
\forall x, y : \mathbb{Z}, \quad & max_2\,(x, y) \geq x \wedge max_2\,(x, y) \geq y \\
& \wedge (max_2\,(x, y) = x \vee max_2\,(x, y) = y)
\end{aligned}
$$

- 期望答案：ite (x <= y) y x

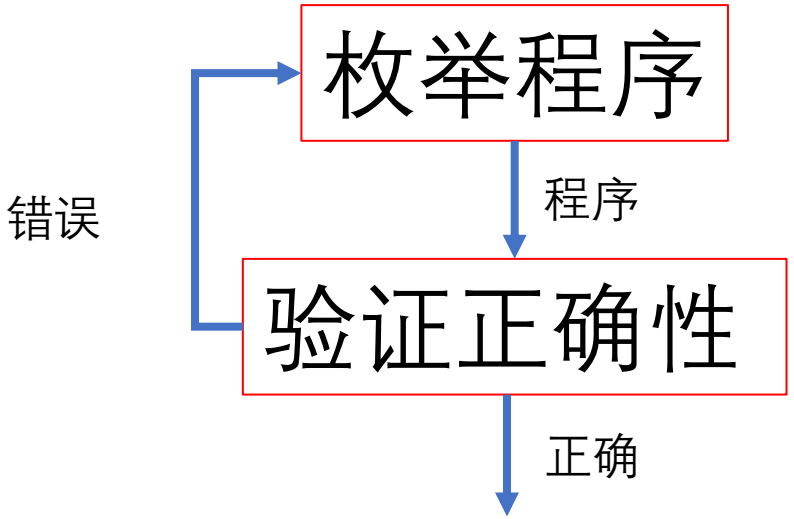
如何求解提升问题?

- 提升问题是程序合成问题
  - 输入:
    - 规约
      - $orig'(op(c, a_1, \ldots, a_n)) = comb(c, orig'(a_1), \ldots, orig'(a_n))$

        $where\ orig'(a) = (orig(a), aux(a))$
    - $comb$的语法
    - aux的语法
  - 输出:
    - $comb$和aux的实现

如何求解程序合成问题

- 程序合成基本方法：枚举+验证



从小到大枚举程序：
x, y, x+y, x+x, y+y, (x+y)+x, (x+y)+y, ...

通过约束求解工具自动验证或者
通过大量测试概率验证

- 其他高级算法可以看做是这个基本形式的优化

程序合成的瓶颈：可伸缩性

- 从小到大的枚举不可能枚举到很大的程序
  - 程序空间随程序大小呈指数增长

$$
\begin{aligned}
&aux(xs) = min(xs)\\
&comb\big((\mathrm{snd}_1, \mathrm{fst}_1), (snd_2, fst_2)\big)\\
&= (\min(snd_1, snd_2, \max(fst_1, fst_2)), \min(fst_1, fst_2))
\end{aligned}
$$

- 当目标程序使用超过5个运算符的时候，现代程序合成工具就经常失效

```
def aux(x):
  mps = max([sum(x[:i+1]) for i in range(lens(x))])
  mts = max([sum(x[i:]) for i in range(lens(x))])
  sumx = sum(x)
  return (mps, mts, sumx)
def comb(L, R):
  (mssL, (mpsL, mtsL, sumL)) = L
  (mssR, (mpsr, mtsR, sumR)) = R
  mss = max(mssL, mssR, mtsL + mpsR)
  mps = max(mpsL, sumL + mpsR)
  mts = max(mtsL + sumR, mtsR)
  sumx = sumL + sumR
  return (mss, (mps, mts, sumx))
```

- 提升问题的解远远超过这个规模

应对可伸缩性问题

- 分治：把解程序分解成小部分单独合成
  - $aux(xs) = \boxed{min(xs)}$
  - $comb\big((\mathrm{snd}_1, \mathrm{fst}_1), (snd_2, fst_2)\big)$

    $= \boxed{(\min(snd_1, snd_2, \max(fst_1, fst_2))}, \boxed{\min(fst_1, fst_2))}$
- 两套分解方法
  - 变量消除
  - 组件消除

组件消除

$$sndmin'(xs_L + xs_R) = comb\ (sndmin'(xs_L), sndmin'(xs_R)) \text{ where } sndmin'(xs) = (sndmin\ xs, aux\ xs)$$

- $(aux, comb)$ 可以分成两部分
  - $aux(xs) = min(xs)$
  - $comb\big((snd_1, fst_1), (snd_2, fst_2)\big) = (\min(snd_1, snd_2, \max(fst_1, fst_2)), \min(fst_1, fst_2))$

1. 提供原来的输出

2. 提供辅助值

- 第一部分的规约
  - $sndmin(xs_L + xs_R) = comb_1\ (sndmin'(xs_L), sndmin'(xs_R))$ where $sndmin'(xs) = (sndmin\ xs, aux\ xs)$
- 求解后，代入原规约得到只有第二部分的规约
  - $aux(xs_L + xs_R) = comb_2\ (sndmin'(xs_L), sndmin'(xs_R))$ where $sndmin'(xs) = (sndmin\ xs, aux\ xs)$

变量消除

$$sndmin\ (xs_L + xs_R) = comb_1\ (sndmin'(xs_L), sndmin'(xs_R))$$
$$\text{where } sndmin'(xs) = (sndmin\ xs, aux\ xs)$$

- 规约中同时包括aux 和 $comb_1$，无法单独合成
- 能否得到只包含aux的规约?

变量消除

$$sndmin\ (xs_L + xs_R) = comb_1\ (sndmin'(xs_L), sndmin'(xs_R))$$
$$\text{where } sndmin'(xs) = (sndmin\ xs, aux\ xs)$$

- $comb_1$ 是一个函数
  - 即同样的输入得到同样的输出

    $$sndmin'\ (xs_L) = sndmin'(xs'_L) \wedge sndmin'(xs_R) = sndmin'(xs'_R)$$
    $$\rightarrow sndmin(xs_L + xs_R) = sndmin(xs'_L + xs'_R)$$
  - 变形可得

    $$sndmin\ (xs_L + xs_R) \neq sndmin\ (xs'_L + xs'_R)$$
    $$\wedge sndmin\ (xs_L) = sndmin\ (xs'_L) \wedge sndmin\ (xs_R) = sndmin\ (xs'_R)$$
    $$\Longrightarrow aux(xs_L) \neq aux(xs'_L) \vee aux(xs_R) \neq aux(xs'_R)$$
  - 该规约只包含aux，可以用传统程序合成技术求解得到aux

消除规则的性质

- 通过上述方法获取的规约是一个近似
  - 如，在变量消除中，只保证存在一个函数 $comb_1$
  - 但该函数不一定能用目标语言实现出程序
  - 在这种情况下，第二步合成$comb_1$ 会失败，需要回溯
  - 回溯过多会影响效率
- 我们证明几乎不会出现回溯

通用情况

- 产生辅助值时可能需要用到更多的辅助值
- 反复应用组件消除，直到没有更多的辅助值产生
- 完整算法请见论文

分治类算法效果

- 96个已有数据集、论文和codeforces.com中的分治和其他问题
  - 最大子段和
  - 字符串转换成数字
  - 检查字符串中括号是否匹配
  - 线段树问题
  - Petrozavodsk冬令营题目（全球243支队伍只有26支解出）

Solver	Total		
	#Solved	T_{Base}	T_{Ours}
AutoLifter	**82**/96	6.53	
Enum	19/96	11.7	**0.14**
Relish	38/96	21.5	**5.33**

动态规划合成效果

- 40个问题，取自Leetcode使用频率最高的动态规划问题和近十年NOIP中的动态规划问题
  - 背包问题
  - 最长公共子序列
  - 最大独立集
  - 网格上的最短路径
  - 斐波那契数列

	#Solved	#Failed (runtime > 1h)	#Best	AvgTime
Our approach	37	3	31	1.87s
SKETCH	0	40	0	–
FOSYNTH	6	34	6	2.64s

算法数据集ASAC [进行中]

- 包含CSP/NOIP近十年左右的比赛题目
- 所有题目用MiniZinc形式化描述
  - 可以直接采用现有CSP/OMT求解器求解
- 所有题目同时包含自然语言描述、测试用例

小结

算法合成是重要的

- 各种软件都需要大量算法

算法设计是困难的

- 研究只是整理出算法设计模式
- 能否应用全凭程序员能力

能否自动应用算法设计模式？

- 初步研究已经在一大类算法模式上成功
- 算法合成的一个通用问题：提升问题
- 通过寻找合适的近似，提升问题可通过分治高效求解

期待更多人投入
算法合成的研究
和实践！

AiDD AI+ 软件研发数字峰会
AI+ software Development Digital summit

感 谢 聆 听