《Stochastic Dynamic Programming Heuristic for the (R, s, S) Policy Parameters Computation》复现与探究

冯戳 20338016
fengj69@mail2.sysu.edu.cn

范雨萱 20338015
fanyx25@mail2.sysu.edu.cn

凌源飞 19331088
lingyf5@mail2.sysu.edu.cn

洪楚轩 20338027
hongchx5@mail2.sysu.edu.cn

Abstract

本文主要研究了非平稳随机问题中的 (R, s, S) 策略参数计算, 并提出了一种基于随机动态规划的启发式算法. 通过复现原论文的结果, 并对算法进行探究和改进, 本文旨在提供对该问题的深入理解和改进思路.

本文首先给出了研究背景和问题描述. 非平稳随机问题是一类具有不确定性的决策问题, 其中 (R, s, S) 策略是一种常用的库存管理策略. 在现有方法部分, 本文对相关的研究方法进行了综述. 其中包括了经典的 (s, S) 问题和 (R, s, S) 问题, 以及 BnB 算法的应用. 通过对现有方法的分析, 本文指出了其在解决非平稳随机问题中的局限性, 为引出新的启发式算法奠定了基础. 接下来, 本文详细介绍了启发式算法的基本思想. 通过对问题的分析和建模, 本文介绍了 $RsS\text{-}SDP$ 算法和 $RsS\text{-}Kconv$ 算法. 这两种算法基于随机动态规划的思想, 通过迭代计算来求解 (R, s, S) 策略的参数. 本文对这两种算法进行了广泛的计算研究, 评估了其质量和计算量.

在结果复现部分, 本文详细介绍了对原论文中提出的启发式算法进行的实验和分析. 首先, 本文评估了不同策略所需的计算时间, 在一个不断增长的时间范围内进行比较. 其次, 在不同需求模式和成本参数下, 对 $RsS\text{-}Kconv$ 算法的表现进行了分析. 最后, 在一个实例中计算了近似最优补货策略, 并对结果进行了进一步的探究和完善.

通过实验结果的分析, 本文发现基于 K-凸性质的松弛算法在计算最优策略时具有较大的优势. 相比于传统的 (R, s, S) 策略计算方法它能够大大减少计算量, 并在 10 或 15 个周期实例中获得优于 $RsS - SDP$ 的结果. 这表明基于 K-凸性质的松弛算法在解决非平稳随机问题中具有较好的性能.

此外, 本文还对原论文中提出的启发式算法进行了改进. 通过对算法的分析和实验结果的比较, 本文基于折半查找的思想提出了一种改进思想, 并对其进行了数值实验. 实验结果表明, 改进算法在计算近似最优补货策略时能够取得较好的效果, 进一步提高了算法的性能和准确性.

关键词: 库存理论, (R, s, S), 动态规划 (SDP), BnB, K-凸性, 折半查找

目录

1 介绍

1.1 背景

存储某种物品的需要是普遍存在的, 商店为满足顾客的需要, 必须有适当数量的库存货物来支持经营, 一旦商品缺货就会损失营业额. 然而过高的库存占用大量流动资金, 需要大量的行政管理与库存费用, 对于经营是不利的. 因此需要一个科学的方法来处理合适补充库存以及补充数量多少的问题, 因此而产生了库存理论.

库存理论的模型主要分为确定性库存模型及随机性库存模型, 确定性库存模型是指需求率恒为常数且时间变量为连续变量, 随机性库存模型是指需求量是服从一定分布规律的随机变量且时间变量以库存周期作为最小单位, 需求的随机性允许模拟真实世界问题的不确定性. 在随机性库存模型中有多种经典策略: (s, S) 型策略是一种动态策略, 即在每个时期检查库存水平, 当库存量低于 s 时立即补货至 S, (R, S) 型策略是一种静态-动态策略, 即在开始时确定补货时间, 在每 R 个周期后均将库存补货至 S , (R, s, S) 是两种策略的推广, 即每 R 个周期后检查库存水平, 当库存量低于 s 时立即补货至 S , (R, s, S) 更通用且有更好的成本性能, 是目前被应用最广泛的策略.

原论文研究库存理论中的重要分支--非平稳随机问题, 计算 (R, s, S) 策略的解. 对于此类问题, 许多学者提出了不断改进的动态规划方法来进行求解: **Scarf** 在 1995 年使用 K-凸性质来对 SDP 的遍历进行优化; **Visentin** 等人在 2021 年提出 $BnB - SDP$ 方法, 利用分枝定界对 SDP 进行优化. 对于模型的求解, 原论文也对动态规划进行了改进, 在 **Scarf** 提出的 SDP 算法的基础上提出了启发性的 $RsS\text{-}SDP$ 算法及 $RsS\text{-}Kconv$ 算法, 并与 **Visentin** 等人所提出的 BnB 算法进行比较. 在 (R, s, S) 策略中, 由基本公式得到最优策略需要非常大量的计算, 而基于 K-凸性质的松弛算法将检查周期单独考虑, 大大减少了最优策略的计算量, 在 10 或 20 个周期实例中都得到了优于 $RsS\text{-}SDP$ 和 BnB 算法的结果.

本文对原论文中提到的问题进行了总结和抽象, 对论文中提到的算法进行了整理和复现, 对比该论文与其他论文中算法的优劣, 并对结果进行进一步的探究与完善, 提出改进算法.

1.2 问题描述

我们考虑在一个 T 期的计划范围内的单商品, 单库存地点的随机库存模型. 为了实现利益的最大化, 我们希望找到一个最优策略 (R, s, S) . 策略 (R, s, S) 表示库存水平会在每隔 R 个时期进行检查, 如果库存水平低于订货点 s , 则会进行补货使库存水平达到最大库存水平 S . 在非稳态随机问题的结构中, 策略参数会随着时间发生改变, 因此我们假设在整个计划范围内的最优策略形式为 $(\mathbf{R}, \mathbf{s}, \mathbf{S})$.

最优策略 $(\mathbf{R}, \mathbf{s}, \mathbf{S})$ 包含了三个方面的库存管理: 每次检查的时间、是否进行补货以及补货数量. 检查的时刻在时间线的一开始就被固定, 并且每次只有在检查之后才能进行补货. 我们将计算最优策略 $(\mathbf{R}, \mathbf{s}, \mathbf{S})$ 等价为最小化整个计划范围内的期望成本.

1.3 假设与符号

推导过程中的基本假设如下

- 检查的时刻在时间线的一开始就被固定
- 只有在检查之后才能进行补货
- 补货在每个时期的开始时完成
- 补货与到货之间没有延迟

- 补货的单位线性成本为 0
- 每个时期的需求量彼此独立且服从某种已知分布

符号	含义
T	计划范围内总共的周期数
R_t	以 t 时期为起点的检查周期的长度
S_t	以 t 时期为起点的检查周期的最大库存水平
s_t	以 t 时期为起点的检查周期的订货点
Q_t	以 t 时期为起点的检查周期的补货量
W	表示每次检查的固定成本
K	表示每次补货的固定成本
h	货物保存到下一时期的单位保管费
b	超出库存水平的需求量的单位缺货损失费
γ_t	0-1 变量, 表示在第 t 时期是否检查
$d_{t,t+i}$	从 t 时期到 $t+i$ 时期的需求量
μ_t	第 t 个时期的需求的分布的均值
I_{t-1}	$t-1$ 时期结束时的库存水平
$f_t(I_{t-1}, Q_t, R_t)$	以 I_{t-1} 的库存水平开始 t 时期到 $t+Rt$ 时期的检查周期的期望成本
C_t	以 I_{t-1} 的库存水平开始 t 时期直到结束的总期望成本

1.4 文章结构

本文结构如下: 第二节对现有方法的理论和实践进行介绍及总结, 其中包括 (s,S) 问题、(R,s,S) 问题及 BnB 算法; 第三节介绍了原文中作者提出的启发式思想; 第四节对 K-凸性的理论及运用进行了详细说明; 第五节使用 Python 对文章结果进行复现; 第六节我们在原文的算法基础上提出了改进思想并进行数值实验, 得到了效果更好的改进算法.

2 现有方法

2.1 (s,S) 问题

为解决 (R,s,S) 问题, 先讨论将 $\mathbf{R}$ 固定后的情况, 即化为 (s,S) 问题. 定义补货量 Q_t 如下:

$$Q_t \triangleq \begin{cases} S_t - I_{t-1} & , \quad I_{t-1} < s_t \\ 0 & , \quad I_{t-1} \geq s_t \end{cases}$$

这表示当上一时期结束时的库存水平 I_{t-1} 小于订货点 s_t 时, 将库存补充至 S_t , 否则不进行补货.

再定义 t 时期的期望成本 f_t 如下:

$$\begin{aligned} f_t(I_{t-1}, Q_t) \triangleq & K\mathbb{I}\{Q_t > 0\} + W \\ & + E\Big[hmax(I_{t-1} - d_{t,t+1} + Q_t, 0) + bmax(-I_{t-1} - Q_t + d_{t,t+1}, 0)\Big] \end{aligned} \tag{2.1}$$

它表示以 I_{t-1} 的库存水平开始 t 时期的期望成本, 其中包括补货成本 (K), 检查成本 (W), 单位保管费 (h), 单位缺货损失费 (b).

用 C_t 表示以 I_{t-1} 的库存水平开始 t 时期直到结束的总期望成本, 那么问题的动态规划方程可以写为：

$$C_t(I_{t-1}) \triangleq \min_{s,S} f_t(I_{t-1}, Q_t) + E[C_{t+1}(I_{t-1} + Q_t - d_{t,t+1})]$$

其目标函数以及边界条件如下：

$$\begin{cases} C_1(I_0) \triangleq \min_{s,S}\{f_1(I_0, Q_1) + E[C_2(I_0 + Q_1 - d_{1,2})]\} \\ C_{T+1}(I_T) \triangleq 0 \end{cases}$$

C_1 表示以 I_0 为初始库存, 从开始到结束的总成本.

按照定义,s 是判断是否补货的极限值, 也就是说, 在所有需要补货的库存量 i 中, 最大的 i 值即为 s 的值. 为了判断库存量 i 是否需要补货, 我们需要比较 $q > 1$ 时,$min\{f_t(i,q) + E(C_{t+1}(i+q-d))\}$ 与 $f_t(i,0) + E(C_{t+1}(i-d))$ 的大小, 若前者小于后者, 则我们认为要补货, 并且对应最小值的 q 即为我们需要的补货量. 而 S 是补货的阈值, 故固定某一个需要补货的库存量 i, 得到其对应的补货量 q 即可算出 $S = i + q$. 我们必须强调 S 的定义与 i 的选取无关, 注意到当不计线性成本时,$f_t(i,q) + E(C_{t+1}(i+q-d))$ 是关于 $i+q$ 的函数, 因此, 对任意需要补货的库存量 I_n, 一定有对应的 q_n, 使得 $I_n + q_n = S$ 恒成立, 并且这一组 (I_n, q_n) 取到 $f_t(i,q) + E(C_{t+1}(i+q-d))$ 的最小值.

为了简化代码, 用 ζ 表示随机变量 $d_{t,t+1}$ 的实现值, $P(d_{t,t+1} = \zeta)$ 表示取该值的概率. 并用算法 2 依据方程 (2.1) 单独计算 f_t .

Algorithm 1: $sS - SDP$

```
for t = T : −1 : 1 do
    cost_best = ∞
    for i = inv_min : 1 : inv_max do
        C_0 = f_t(i, 0) + E[C_{t+1}(i − d_{t,t+1})]
        Ĉ_t(i) = C_0
        for q = 1 : 1 : q_max do
            if f_t(i, q) + E[C_{t+1}(i + q − d_{t,t+1})] < Ĉ_t(i) then
                Ĉ_t(i) = f_t(i, q) + E[C_{t+1}(i + q − d_{t,t+1})]
            end
            if Ĉ_t(i) ≤ cost_best and Ĉ_t(i) < C_0 then
                cost_best = Ĉ_t(i)
                opt_i = i
                opt_q = q
            end
        end
    end
    C_t = Ĉ_t
    s_t = opt_i
    S_t = opt_i + opt_q
end
```

Algorithm 2: $f_t(i,q)$

```
cost = W
if q > 0 then
    cost = cost + K
end
for each ζ value of d_{t,t+1} do
    opt_inv = i + q − ζ
    if opt_inv ≥ 0 then
        cost = cost + h opt_inv P(d_{t,t+1} = ζ)
    else
        cost = cost − b opt_inv P(d_{t,t+1} = ζ)
    end
end
```

2.2 (R,s,S) 问题

类比 (s,S) 问题, 加入变量 R 后, 更新 Q_t, f_t 的表达式, 其中 f_t 表示以 I_{t-1} 的库存水平开始 t 时期到 $t+R_t$ 时期的检查周期的期望成本.

$$Q_t \triangleq \begin{cases} S_t - I_{t-1} & , \ I_{t-1} < s_t \\ 0 & \text{其他} \end{cases}$$

$$\begin{aligned} f_t(I_{t-1}, Q_t, R_t) \triangleq\ & K\mathbb{I}\{Q_t > 0\} + W \\ & + \sum_{i=1}^{R_t} E\Big[hmax(I_{t-1} - d_{t,t+i} + Q_t, 0) + bmax(-I_{t-1} - Q_t + d_{t,t+i}, 0)\Big] \end{aligned} \tag{2.2}$$

同理用 C_t 表示以 I_{t-1} 的库存水平开始 t 时期直到结束的总期望成本, 它等于 t 时期到 $t+R_t$ 时期的检查周期的期望成本加上 C_{t+R_t}

$$C_t(I_{t-1}) \triangleq \min_{R,s,S} f_t(I_{t-1}, Q_t, R_t) + E[C_{t+R_t}(I_{t-1} + Q_t - d_{t,t+R_t})] \tag{2.3}$$

类似地可以得到动态规划的目标函数及初始条件:

$$\begin{cases} C_1(I_0) \triangleq \min\limits_{R,s,S} f_1(I_0, Q_1, R_1) + E[C_{1+R_1}(I_0 + Q_1 - d_{1,1+R_1})] \\ C_{T+1}(I_T) \triangleq 0 \end{cases}$$

直接计算该动态规划方程较复杂, 该算法计算量会随着 T 的增加而呈指数型增长, 无法大规模使用, 因此以下考虑 SDP 的改进方法.

2.3 BnB

引入审查轮次 $\boldsymbol{\gamma} = (\gamma_1, \gamma_2, \cdots, \gamma_T)$, 其中 $\gamma_t = 1$ 表示 t 时期开始时进行检查, $\gamma_t = 0$ 表示 t 时期开始时不进行检查.

将方程 (2.2), (2.3) 改写成如下形式:

$$\begin{aligned} f_t(I_{t-1}, Q_t) \triangleq\ & K\mathbb{I}\{Q_t > 0\} + \gamma_t W \\ & + E\Big[hmax(I_{t-1} - d_{t,t+1} + Q_t, 0) + bmax(-I_{t-1} - Q_t + d_{t,t+1}, 0)\Big] \\ C_t(I_{t-1}) \triangleq\ & \min_{0 \le Q_t \le M\gamma_t} \big(f_t(I_{t-1}, Q_t) + E[C_{t+1}(I_{t-1} + Q_t - d_{t,t+1})]\big) \end{aligned}$$

在 SDP 中对 R_t 进行遍历等价于对 γ_t 遍历, 由于 γ_t 的取值只能为 0 或 1 , 因此可以用二叉树的形式表示对 γ_t 遍历的过程. 在遍历的过程中可以根据已经求出的最小成本排除一些情况, 类似对树进行剪枝.

根据此思想, 可以先写出目标函数并引入如下定理及推论:

$$\hat{C_1}(I_0) = \min_{\gamma_1,\ldots,\gamma_T}(C_1(I_0))$$

定理 1. *对于固定的审查轮次 $\boldsymbol{\gamma}$, 有如下不等式成立:*

$$\underset{I_{t-2}}{min}\, C_{t-1}(I_{t-2}) \geq \underset{I_{t-1}}{min}\, C_t(I_{t-1})$$

由定义易知其成立, 以此有如下推论:

推论 1. *给定任意常数 $N > 0$, 若存在某个 $t > 1$, 使得 $\underset{I_{t-1}}{min}\, C_t(I_{t-1}) \geq N$, 则必有 $C_1(I_0) \geq N$.*

尝试对所有的审查轮次的组合进行筛选. 设 $\boldsymbol{\gamma_i} = (\gamma_{i,1},\gamma_{i,2},\cdots,\gamma_{i,T})$ 为第 i 次遍历时的审查轮次. 假设已经得到了目前期望成本的一个上界 $\hat{C}$. 则在遍历时, 若存在某一个 $t_1 > 0$, 使得: $\underset{I_{t_1-1}}{min}\, C_{t_1}(I_{t_1-1},\boldsymbol{\gamma}) \geq \hat{C}$, 则由**推论一**, 必有:

$$C_1(I_0,\boldsymbol{\gamma}) \geq \hat{C}$$

因此, 当我们算到上述的时刻 t 的时候, 其实已经不必计算剩下的 C_{t_1-1} 到 C_1 , 由此可以把这一次遍历中止, 这就类似分支定界法中的“剪枝”过程. 继续上述过程, 如果在第 n 次遍历中有 $C_1(I_0,\boldsymbol{\gamma_n}) < \hat{C}$ 成立, 那么更新上界 $\hat{C} = C_1(I_0,\boldsymbol{\gamma_n})$, 并继续上述过程, 直到所有 $\boldsymbol{\gamma_i}$ 均遍历完毕.

以上方法即称为对 (R,s,S) 问题的 **分支定界法**.

但实际上, 该方法在大多数情况下只能修剪很少一部分的枝条. 其根本原因在于, $\hat{C}$ 已经计算了在固定某个审查轮次时, 时期 1 到 T 的一个最小成本. 但我们在比较上界时只用到了形如 $\underset{I_{t-1}}{min}\, C_t(I_{t-1},\boldsymbol{\gamma_n})$ 的值去比较, 换而言之, 我们缺失了时期 1 到 t 的影响, 这使得满足修剪枝条不等式的情况并不总是多见. 为了解决这个问题, 我们引入 $MC_t(I_t)$ 用于弥补计算时缺失的时期 1 到 t 的成本. $MC_t(I_t)$ 表示当 t 时期的最终库存为 I_t 时, 时期 1 到时期 t 的最小成本:

$$MC_1(I_1) = \begin{cases} f_1(I_0, I_1 - I_0 + min(d_1), 1), & I_1 > I_0 - min(d_1) \\ f_1(I_0, 0, 0), & I_1 \leq I_0 - min(d_1) \end{cases}$$

其中 $f_1(I_0, I_1 - I_0 + min(d_1), 1)$ 表示时刻 1 检查时的最小成本, $f_1(I_0, 0, 0)$ 表示不检查时的最小成本. 当 $I_1 > I_0 - min(d_1)$ 时, 需要补货, 当需求量最小时补货量最少, 进而保证取到检查时的最小成本. 同理, 在 $I_1 \leq I_0 - min(d_1)$ 时, 最小成本在不检查的时候取到. 类似地, 有如下递归式:

$$MC_t(I_t) = min \begin{cases} \underset{j<I_t+min(d_t)}{min}(f_t(j, I_t - j + min(d_t), 1) + MC_{t-1}(j)) \\ \underset{j\geq I_t+min(d_t)}{min}(f_t(j, 0, 0) + MC_{t-1}(j)) \end{cases}$$

定义了 $MC_t(I_t)$ 后, 可进一步减小分支定界法的“修剪”上界, 更新“修剪”条件如下:

$$\underset{I_{t_1-1}}{min}\, C_{t_1}(I_{t_1-1},\boldsymbol{\gamma}) \geq \hat{C} \Rightarrow \min_{I_t}(C_t(I_{t-1},\boldsymbol{\gamma}) + MC_{t-1}(I_{t-1})) \geq \hat{C} \tag{2.19}$$

算法的伪代码如下

Algorithm 3: $BnB - SDP(t, [\gamma_t, \ldots, \gamma_T])$

if $t == T$ **then** $\hat{C} = \infty$;
if *t=1* **then**
  if $r_t == 1$ **then**
    $cost_{min} = \infty$
    for $q = 1 : 1 : q_{max}$ **do**
      if $f_t(I_0, q) + E[C_{t+1}(I_0 + q - d_{t,t+1})] < \hat{C}$ **then**
        $\hat{C} = f_t(I_0, q) + E[C_{t+1}(I_0 + q - d_{t,t+1})]$
      end
      for $i = inv_{min} : 1 : inv_{max}$ **do**
        if $f_t(i, q) + E[C_{t+1}(i + q - d_{t,t+1})] < cost_{min}$ **then**
          $cost_{min} = f_t(i, q) + E[C_{t+1}(i + q - d_{t,t+1})]$
          $s_t = i$
          $S_t = i + q$
        end
      end
    end
  else if $f_t(I_0, q) + E[C_{t+1}(I_0 + q - d_{t,t+1})] < \hat{C}$ **then**
    $\hat{C} = f_t(I_0, q) + E[C_{t+1}(I_0 + q - d_{t,t+1})]$
  end
else
  if $r_t = 1$ **then**
    $cost_{best} = \infty$
    for $i = inv_{min} : 1 : inv_{max}$ **do**
      $C_t(i) = \infty$
      for $q = 1 : 1 : q_{max}$ **do**
        if $f_t(i, q) + E[C_{t+1}(i + q - d_{t,t+1})] < C_t(i)$ **then**
          $C_t(i) = f_t(i, q) + E[C_{t+1}(i + q - d_{t,t+1})]$
          if $C_t(i) \leq cost_{best}$ **then**
            $cost_{best} = C_t(i)$
            $s_t = i$
            $S_t = i + q$
          end
        end
      end
    end
  else
    for $i = inv_{min} : 1 : inv_{max}$ **do** $C_t(i) = f_t(i, 0) + E[C_{t+1}(i - d_{t,t+1})]$;
  end
  if $min(C_t(i) + MC_t(i)) \geq \hat{C}$ **then break**;
  else if $min(C_t(i) + MC_t(i)) > \hat{C}$ **then**
    $BnB - SDP(t - 1, [0, \gamma_t, \ldots, \gamma_T])$
    $BnB - SDP(t - 1, [1, \gamma_t, \ldots, \gamma_T])$
  end
end

Algorithm 4: $MC_t(i)$

```
if t == 0 then
    MC = 0
    return MC
end
MC = ∞
for j = 1 : 1 : i + min(d_t) − 1 do
    opt = W + K + h max{j, 0} − b max{j, 0} + MC_{t−1}(j)
    if opt < MC then
        MC = opt
    end
end
for j = i + min(d_t) : 1 : inv_max do
    opt = W + K + h max{j, 0} − b max{j, 0}
    if opt < MC then
        MC = opt
    end
end
return MC
```

3 启发式算法

3.1 基本思想

传统的 SDP 算法中, 通过不断迭代将所有变量集合于一个目标函数中, 之后只能通过遍历所有变量取值的方式求解最优策略. BnB 算法通过剪枝可一定程度上减少遍历的次数, 但由于算法本身计算量非常大, 剪枝后仍有大量取值需遍历, 该算法对计算量的减少是有限的. 因此作者提出启发法, 进一步减少计算量.

在 (R, s, S) 问题中, 若能先固定 R , 将问题简化为 (s, S) 问题, 则可以减少遍历的变量个数, 进而减少计算量. 在求解 R 时, 启发法的主要思想是使用类似贪心算法的思想, 从 T 开始向前计算得到每一步的局部最优 R_t , 在已计算出的 R_t 基础上计算局部最优 R_{t-1} , 而非传统算法中将所有 R_t 集中至目标函数中一次求解. 基于此思想, 可以将方程 (2.3) 改为以下形式:

$$\hat{C}_t(I_{t-1}) = \min_{R_t} f_t(I_{t-1}, Q_t, R_t) + E[C_{t+R_t}(I_{t-1} + Q_t - d_{t,t+R_t})]$$

把求解 (R, s, S) 拆分为三个步骤:

1. 先遍历 I_{t-1} 和 Q_t 的值, 求出 $arg\min_{I_{t-1}} \hat{C}_t(I_{t-1})$

$$\tilde{I}_t \triangleq arg\min_{I_{t-1}} \hat{C}_t(I_{t-1})$$

2. 再遍历 R_t , 求出

$$R_t^a \triangleq arg\min_{R_t} \hat{C}_t(\tilde{I}_t) \tag{3.1}$$

3. 将近似得到的 R_t^a 代入动态规划方程, 将原问题简化为 (s, S) 问题, 可以求解近似期望成本, 从而制定近似最优策略:

$$C_t^a(I_{t-1}) \triangleq f_t(I_{t-1}, Q_t, R_t^a) + E[C_{t+R_t^a}^a(I_{t-1} + Q_t - d_{t,t+R_t^a})] \tag{3.2}$$

3.2 伪代码

展示了回溯启发法的程序. 第 1-2 行设定了所有库存水平下的边界条件. 第 4 行以倒序的方式遍历所有的时期. 第 6 行遍历所有可能的检查周期, 第 7 行遍历所有的库存水平, 第 9 行遍历所有可能的补货量, 第 11 行根据方程 (3.2) 计算近似期望成本. 第 16-17 行根据方程 (3.1) 计算 R_t^a 的当前值, 而第 18 行选取最小近似期望成本.

为了简化代码, 用 $\zeta_{t,t+j}$ 表示随机变量 $d_{t,t+j}$ 的实现值, $P(d_{t,t+j} = \zeta_{t,t+j})$ 表示取该值的概率. 并用算法 6 依据方程 (2.2) 单独计算 f_t .

Algorithm 5: *RsS-SDP*

```
1  for i = inv_min : 1 : inv_max do
2  |  C^a_{T+1} = 0
3  end
4  for t = T : −1 : 1 do
5  |  cost_best = ∞
6  |  for r = 1 : 1 : T − t + 1 do
7  |  |  for i = inv_min : 1 : inv_max do
8  |  |  |  Ĉ(i) = ∞
9  |  |  |  for q = 0 : 1 : q_max do
10 |  |  |  |  if f_t(i, q, r) + E[C^a_{t+r}(i + q − d_{t,t+r})] < Ĉ(i) then
11 |  |  |  |  |  Ĉ(i) = f_t(i, q, r) + E[C^a_{t+r}(i + q − d_{t,t+r})]
12 |  |  |  |  end
13 |  |  |  end
14 |  |  end
15 |  |  if min{Ĉ(i)} < cost_best then
16 |  |  |  cost_best = min{Ĉ(i)}
17 |  |  |  R^a_t = r
18 |  |  |  C^a_t = Ĉ
19 |  |  end
20 |  end
21 end
```

Algorithm 6: $f_t(i,q,r)$

if $r == 0$ **then**
  return 0
end
$cost = W$
if $q > 0$ **then**
  $cost = cost + K$
end
for $j = 1:1:r$ **do**
  for *each* $\zeta_{t,t+j}$ *value of* $d_{t,t+j}$ **do**
    $\text{opt}_{inv} = i + q - \zeta_{t,t+j}$
    if $\text{opt}_{inv} \geq 0$ **then**
      $cost = cost + h\,\text{opt}_{inv}\,P(d_{t,t+j} = \zeta_{t,t+j})$
    else
      $cost = cost - b\,\text{opt}_{inv}\,P(d_{t,t+j} = \zeta_{t,t+j})$
    end
  end
end

然而该方法得到的近似 SDP 公式在计算时仍需遍历当前库存水平 I_{t-1} , 补货量 Q_t 以及检查周期长度 R_t^a , 计算量令人望而却步. 在 (s,S) 策略中, 计算量可以通过 K-凸性大量减少, 作者类比该性质并将其应用于 (R,s,S) 策略中, 得到新的计算方法.

4 K-凸性

在 **Scarf(1959)** 中作者介绍了 K-凸性, 我们可以运用该性质对以上算法进一步简化.

4.1 理论基础

4.1.1 定义与性质

定义. 对于 $K \geq 0$, 称函数 $f(x)$ 为 K-凸函数, 若对于任意的 $a,b > 0$, 任意的 x , 有

$$K + f(a+x) - f(x) - a(\frac{f(x) - f(x-b)}{b}) \geq 0$$

K-凸函数有以下性质:

1. 若 $f(x)$ 是 0-凸函数, 则对任意 $K > 0$, $f(x)$ 是 K-凸函数.
2. 若 $f(x)$ 是 K-凸函数, 则对任意 h , $f(x+h)$ 也是 K-凸函数.
3. 若 f 是 K-凸函数, g 是 M-凸函数, 则对任意 $\alpha,\beta > 0$, $\alpha f + \beta g$ 是 $(\alpha K + \beta M)$-凸函数.
4. 若 $f(x)$ 是 K-凸函数, 且 $F(x) = \sum_{\delta=0}^{\infty} f(x+\delta)p(\delta)d\delta$ 收敛, 其中 $p(\delta)$ 为概率函数, 则 $F(x)$ 也是 K-凸函数.
5. 若 $f(x)$ 是 K-凸函数, 且 $F(x) = \int_0^{\infty} f(x+\delta)\phi(\delta)d\delta$ 收敛, 其中 $\phi(\delta)$ 为密度函数, 则 $F(x)$ 也是 K-凸函数.

4.1.2 性质证明

命题. *若 $f(x)$ 是 K-凸函数, 且 $F(x)=\int_0^\infty f(x+\delta)p(\delta)d\delta$ 收敛, 其中 $p(\delta)$ 为概率函数, 则 $F(x)$ 也是 K-凸函数.*

证明. 由定义出发, 对任意的 a , b 使 $a+x$, $x-b$ 均在定义域内, 有:

$$\begin{aligned}K+F(a+x)-F(x)&=K+\int_0^\infty[f(a+x+\delta)-f(x+\delta)]d(\delta)d\delta\\&\geq K+\int_0^\infty\{a[\frac{f(x+\delta)-f(x-b+\delta)}{b}]-K\}\phi(\delta)d\delta\\&=K\{1-\int_0^\infty\phi(\delta)d\delta)+a[\frac{F(x)-F(x-b)}{b}]\\&\geq a[\frac{F(x)-F(x-b)}{b}]\end{aligned}$$

由此即知 $F(x)$ 是 K-凸的. □

命题. *若 $f(x)$ 是 K-凸函数, 且 $F(x)=\sum_{\delta=0}^\infty f(x+\delta)\phi(\delta)$ 收敛, 其中 $\phi(\delta)$ 为密度函数, 则 $F(x)$ 也是 K-凸函数.*

证明. 由定义出发, 对任意的 a , b 使 $a+x$, $x-b$ 均在定义域内, 有:

$$\begin{aligned}K+F(a+x)-F(x)&=K+\sum_{a=0}^\infty[f(a+x+\delta)-f(x+\delta)]p(\delta)d\delta\\&\geq K+\sum_{a=0}^\infty(a[\frac{f(x+\delta)-f(x-b+\delta)}{b}]-K]P(\delta)d\delta\\&=K(1-\sum_{b=0}^\infty p(\delta)d\delta)+a[\frac{F(x)-F(x-b)}{b}]\\&\geq a[\frac{F(x)-F(x-b)}{b}]\end{aligned}$$

由此即知 $F(x)$ 是 K-凸的. □

4.1.3 相关命题及引理

命题. *对于 K-凸函数 $f(x)$, 定义 $S=min\{x|f(x)=minf(x)\}$, $s=inf\{x\leq S|f(x)<f(S)+K\}$ (s 可能不在定义域内), 有如下性质:*

1. *若 s 在定义域内, 则 $f(x)$ 在区间 $x<s$ 上单调递减.*
2. *若 s 和 S 存在, 且在定义域内, 则对任意 $s<x_1<x_2$, 有 $f(x_2)+K\geq f(x_1)$*

证明. 1. 由 s 的定义, 当 $x<s$ 时, 必有

$$f(x)\geq f(S)+K$$

此时取 $a=S-x>0$, 由 K-凸函数定义, 对任意 $x-b$, $b>0$ 在定义域内, 有:

$$f(x)-f(x-b)\leq\frac{b}{a}[K+f(x+a)-f(x)]=\frac{b}{a}[K+f(S)-f(x)]\leq 0$$

故当 $x<s$ 时, $f(x)$ 是递减的.

2. 若 $x_1>S$, 则令 $x=x_1$, $a=x_2-x_1>0$, $b=x_1-S>0$, 由定义有:

$$K+f(x_2)\geq f(x_1)+a[\frac{f(x_1)-f(S)}{b}]\geq f(x_1)$$

若 $x_2 < S$, 先说明对任意 $s < x < S$, 均有 $f(x) \leq K + f(S)$. 假设不等式不恒成立, 由 s 的定义, 必存在 $s < x' < x$, 使得 $f(x') < K + f(S)$, 故由定义有:

$$K + f(S) \geq f(x) + (S - x)[\frac{f(x) - f(x')}{x - x'}] \geq f(x)$$

与假设矛盾, 故上述不等式恒成立
则对任意 $x_1 < x_2 < S$ 有:

$$K + f(x_2) > K + f(S) \geq f(x_1)$$

若 $x_1 < S \leq x_2$, 同样有:

$$K + f(x_2) \geq K + f(S) \geq f(x_1)$$

证毕. □

引理. ***Scarf (1959)*** 沿用前文的记号如下

$$\begin{aligned} C_t(I_{t-1}) &\triangleq \min f_t(I_{t-1}, Q_t, R_t) + E[C_{t+R_t}(I_{t-1} + Q_t - d_{t,t+R_t})] \\ f_t(I_{t-1}, Q_t, R_t) &\triangleq K\mathbb{I}\{Q_t > 0\} + W \\ &\quad + \sum_{i=1}^{R_t} E\Big[hmax(I_{t-1} - d_{t,t+i} + Q_t, 0) + bmax(-I_{t-1} - Q_t + d_{t,t+i}, 0)\Big] \end{aligned}$$

然后做如下变换, 其中 $y = I_{t-1} + Q_t$, 记

$$\begin{aligned} L_t(y) &= \sum_{i=1}^{R_t} E\Big[hmax(y - d_{t,t+i}, 0) + bmax(-y + d_{t,t+i}, 0)\Big] \\ G_t(y) &= C_t(I_{t-1}) - K\mathbb{I}\{Q_t > 0\} - W = L_t(y) + E[C_{t+R_t}(y - d_{t,t+R_t})] \\ &= L_t(y) + \int_0^\infty C_{t+R_t}(y - \delta)\phi_{R_t}(\delta)d\delta \end{aligned}$$

那么当 $G_t(y)$ 为 K-凸函数时, 存在一组上述的 (s, S) 在定义域内, 其为 (s_t, S_t) 的最优值, 并且有如下等式成立: $G_t(s_0) = G_t(S_0) + K$

4.2 K-凸性的证明和应用

下面证明 C_t 是 K-凸的, 以此简化 (s, S) 问题的计算过程, 首先利用数学归纳法证明 $G_t(y)$ 是 K-凸的:

$$\forall t > 0,\ L_t(y) = \sum_{i=1}^{R_t}[\int_0^y h(y - \delta)\phi_i(\delta)d\delta + \int_y^\infty b(\delta - y)\phi_i(\delta)d\delta]$$

代入 **定义**, 易知 L_t 是 0-凸的.

由初始条件, $G_T = L_T(y)$, 从而 G_T 是 K-凸的. 假设 $G_T, G_{T-1}, \cdots, G_{t+1}$ 是 K-凸的, 下面证明 G_t 是 K-凸的.

对固定的 R_t ,

$$G_t(y) = L_t(y) + \int_0^\infty C_{t+R_t}(y - \delta)\phi_{R_t}(\delta)d\delta$$

由于 L_t 是 0-凸的, 只需要证明 $\int_0^\infty C_{t+R_t}(y-\delta)\phi_{R_t}(\delta)d\delta$ 是 K-凸的即可. 由上述 K-凸函数的性质 *2* 与 *5*, 只需要说明 C_{t+R_t} 的 K-凸性即可.

设 $t + R_t = n$, 由 R_t 的定义有 $t < n < T$, 由假设知 G_n 是 K-凸的, 再由引理知存在唯一的 (s_n, S_n) 为最优策略, 故由定义知

$$C_{t+R_t}(I_{t-1}) = C_n(I_{t-1}) = \begin{cases} W + G_n(I_{t-1}), & s_n \leq I_{t-1} < S_n \\ K + W + G_n(S_n), & 0 < I_{t-1} < s_n \end{cases}$$

(1) $x-b>s_n$ 时

由 G_n 的 K-凸性可得 C_n 是 K-凸的.

(2) $x-b<s_n<x$ 时

$$\begin{aligned}&K+C_n(x+a)-C_n(x)-a[\frac{C_n(x)-C_n(x-b)}{b}]\\=&K+G_n(x+a)-G_n(x)-a[\frac{G_n(x)-G_n(S_n)-K}{b}]\\=&K+G_n(x+a)-G_n(x)-a[\frac{G_n(x)-G_n(s_n)}{b}]\end{aligned}$$

由 **4.1.3 命题.1** 知 $K+G_n(x+a)-G_n(x)\geq 0$

(a) 若 $G_n(x)\leq G_n(s_n)$, 则上式大于 0.

(b) 若 $G_n(x)>G_n(s_n)$, 由于 $x-s_n<b$, 即知:

$$\begin{aligned}&K+G_n(x+a)-G_n(x)-a[\frac{G_n(x)-G_n(s_n)}{b}]\\>&K+G_n(x+a)-G_n(x)-a[\frac{G_n(x)-G_n(s_n)}{x-s_n}]\end{aligned}$$

由 G_n 的 K-凸性知上式大于 0 .

故该情况下 C_n 为 K-凸的.

(3) $x<s_n<x+a$

此时 $K+C_n(x+a)-C_n(x)-a[\frac{C_n(x)-C_n(x-b)}{b}]=K+C_n(x+a)-C_n(x)$,

$$\begin{aligned}C_n(x)&=\min_{y\geq x}(K1(Q_t>0)+W+L_n(y)+E[C_{n+R_n}(y-d_{n,n+R_n})])\\&\leq K+W+L_n(x+a)+E[C_{n+R_n}(x+a-d_{n,n+R_n})]\\&=K+C_n(x+a)\end{aligned}$$

故 $K+C_n(x+a)-C_n(x)\geq 0$, 即 C_n 是 K-凸的.

(4) $x+a<s_n$, 在这种情况下, $C_n(x-b)=C_n(x)=C_n(x+a)=K+W+G_n(S_n)$ 为常数, C_n 显然是 K-凸的.

综上, C_n 为 K-凸的, 由此得 G_t 是 K-凸的.

故由数学归纳法即知, 对任意 $1\leq t\leq T$, 都有 G_t 为 K-凸函数, 证毕.

根据引理, 可知对任意的 G_t , 都存在唯一一组最优策略 (s_t^*,S_t^*) , 使得下式成立:$G_t(s_t^*)=G_t(S_t^*)+K$, 且有

$$C_t(I_{t-1})=\begin{cases}W+G_t(I_{t-1}), & s_t^*\leq I_{t-1}<S_t^*\\K+W+G_t(S_t^*), & 0<I_{t-1}<s_t^*\end{cases}$$

由 G_t 的 K-凸性即知 C_t 也具有 K-凸性

4.3 伪代码

由 C_t 的 $K-$ 凸性, 我们知道 $C_t(i)$ 的最小值在 $q=0$ 处取得并且唯一. 故第 9 行在遍历成本函数时只需要在 $q=0$ 的情况下遍历所有的 i, 就一定能取得最小值. 因为 S 为 $C_t(i)$ 的最小值点, 故在第 10 行我们将得到的更小的 $C_t(i)$ 所对应的 i 赋值给 S. 当 S 值不再更新后,i 的遍历仍旧继续. 这是为了求出 s 的值. 我们由 s 与 S 之间的等式关系知道,$C_t(s)=C_t(S)+K$ 也成立, 并且对于 $s<x<S$ 之间的值有 $C_t(x)<C_t(S)+K$（这由

$K-$ 凸函数中的性质可知）, 即在 $q=0$ 时, 小于 $C_t(S)+K$ 的成本所对应的 i 值均不为 s. 这告诉我们, 使得成本大于等于 $C_t(S)$ 加上 K 所对应的 i 的最大值即为 s 的值, 故在 14 行我们将满足这个情况的 i 值赋予 s. 并且 i 从大到小遍历, 可以保证取得的值是最大的, 得到此值后需要立刻中止当前循环. 在第 19 行, 如果当前 r 对应的 C_t 最小值小于之前记录的值, 由启发法思想, 需要将 R_t 的值更新为 r. 最后, 在第 25 行, 将 $C_t(i<s)$ 更新为补充到 S 时的成本值.

Algorithm 7: *RsS-SDP-Kconv*

```
1  for i = inv_min : 1 : inv_max do
2  |  C^a_{T+1} = 0
3  end
4  for t = T : −1 : 0 do
5  |  review_cost_best = ∞
6  |  for r = 1 : 1 : T − t + 1 do
7  |  |  cycle_cost_best = ∞
8  |  |  for i = inv_max : −1 : inv_min do
9  |  |  |  Ĉ_t(i) = f_t(i, 0, r) + E[C^a_{t+r}(i − d_{t,t+r})]
10 |  |  |  if Ĉ_t(i) < cycle_cost_best then
11 |  |  |  |  cycle_cost_best = Ĉ_t(i)
12 |  |  |  |  Ŝ_t = i
13 |  |  |  end
14 |  |  |  if Ĉ_t(i) > cycle_cost_best + K then
15 |  |  |  |  ŝ_t = i
16 |  |  |  |  break
17 |  |  |  end
18 |  |  end
19 |  |  if cycle_cost_best < review_cost_best then
20 |  |  |  review_cost_best = cycle_cost_best
21 |  |  |  R^a_t = r
22 |  |  |  C^a_t = Ĉ_t
23 |  |  |  s_t = ŝ_t
24 |  |  |  S_t = Ŝ_t
25 |  |  |  for i = inv_min : 1 : s_t do
26 |  |  |  |  C_t(i) = Ĉ_t(S_t)
27 |  |  |  end
28 |  |  end
29 |  end
30 end
```

5 结果复现

本节对原论文中提出的启发式进行了广泛的计算研究. 我们的目标是评估由启发式计算的策略的质量和所需的计算量. 在第 5.1 节中, 我们评估在一个不断增长的时间范围内, 比较不同策略所需的计算时间; 在 5.2 节中, 我们分析不同需求模式和成本参数下,$RsS\text{-}Kconv$ 算法的表现. 最后在 5.3 节中, 我们分析该算法计算一个近似最优补货策略的实例.

在实验中, 我们使用 **Visentin** 等人 (2021) 提出的 BnB 算法作为比较, 用其求解得到的期望成本作为理论最优值来计算相对最优性差距.

$$\text{相对最优性差距} \triangleq \frac{\text{实际成本} - \text{理论最优成本}}{\text{理论最优成本}}$$

相对最优性差距可用于衡量该策略与理论最优策略的差距, 因此越小的相对最优性差距意味着策略越好.

在评估 $RsS\text{-}Kconv$ 算法的表现时, 使用以下两种方法作为对比:

. BnB, **Visentin** 等人 (2021) 提出的最快的分支定界方法.

. $RsS\text{-}SDP$, 即 Algorithm 5 中提出的 SDP 启发式模型的基本实现.

5.1 算法效率

我们设置参数如下:

$h = 1$, 其他成本参数从均匀随机变量中抽样: $K \in [80, 320]$, $W \in [80, 320]$, $b \in [4, 16]$. 需求量 $d_{t,t+i}$ 服从泊松分布, 均值 μ 从 30 到 70 中均匀抽取, 考虑周期数 $T \in [5, 40]$ 作为横轴, 对每一个 T 生成 100 个实例, 取平均计算时间的对数作为纵轴, 绘制图 1, 用于比较三种算法在计算量上的优劣.

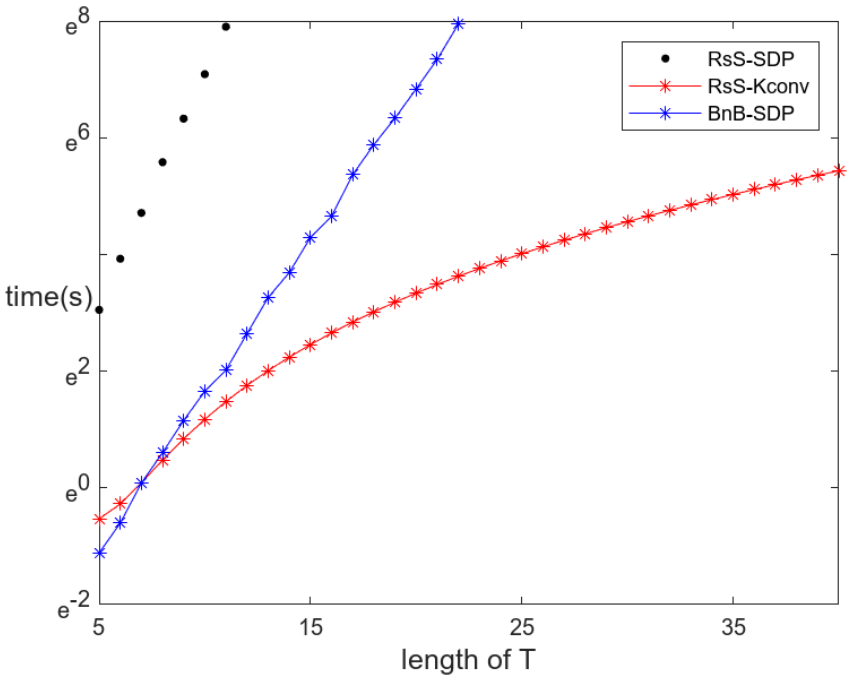

图 1: 三种算法的平均计算时间

由图 1,$RsS\text{-}SDP$ 算法在时间消耗上明显劣于其余两个算法. 相对来说 $RsS\text{-}Kconv$ 算法和 BnB 算法减少了计算时间. 在 7 个周期以内的实例中,BnB 算法的表现略优于 $RsS\text{-}Kconv$ 算法, 在 7 个周期以上时两者之间的差距随周期数增加明显增大.$RsS\text{-}Kconv$ 算法性能在时间上的改善比 BnB 算法性能的改善更为显著.

5.2 不同参数下的算法表现

为了分析在不同实例参数下 $RsS\text{-}Kconv$ 算法的性能, 以及哪些参数影响了该算法的计算性能和相对最优性差距, 我们设置更广泛的检查和补货成本, 并增加需求的不确定性 (通过改变需求量 d 服从的正态分布的参数来实现), 设置如下参数范围:

$$T \in \{10, 15\}, K, W \in \{20, 40, 80, 160, 320\}, h = 1, b = 10$$

需求量 $d_{t,t+i}$ 服从均值为 μ , 方差为 $\sigma\mu$ 的正态分布, 为模拟需求的不同趋势以及波动性, 其中 $\sigma \in \{0.1, 0.2, 0.3, 0.4\}$ 用于模拟需求的不确定性, 0.4 用于模拟极端不确定情形. 设置以下模式 (μ的分布):

. (STA) 平稳趋势:

$$\mu = 50$$

. (INC) 增长趋势:

$$\mu = \frac{100t}{n-1}$$

. (DEC) 下降趋势:

$$\mu = 100 - \frac{100t}{n-1}$$

. (LCY1) 增长-平稳-下降周期波动:

$$\mu(t) = \begin{cases} \frac{(2t+1)75}{2\text{round}(n/3)} & t < \lfloor n/3 \rfloor \\ 75 & \lfloor n/3 \rfloor \le t < 2\lfloor n/3 \rfloor + n\%3 \\ 75 - \frac{(2t-1)75}{2\text{round}(n/3)+1} & 2\lfloor n/3 \rfloor + n\%3 \le t < n \end{cases}$$

. (LCY2) 增长-下降周期波动:

$$\mu(t) = \begin{cases} \frac{(2t+1)\cdot 100}{2\text{round}(n/2)} & t < \lfloor n/2 \rfloor \\ 100 - \frac{(2t+1)\cdot 100}{2\text{round}(n/2+0.5)} & \lfloor n/2 \rfloor \le t < n \end{cases}$$

. (RAND) 随机:

$$\mu \sim U(1, 100)$$

根据以上参数设置, 每一组参数组合生成 10000 个实例, 计算平均相对最优差距、平均计算时间、平均检查次数、成本误差, 得到以下表格:

表 1 和表 2 分别显示了周期数 T 取 10 和 15 的情况. 考虑成本参数, 在高补货成本和低检查成本的情况下, 相对最优性差距较大, 即 $RsS\text{-}Kconv$ 算法无法有效计算出最优策略. 在低检查成本的情况下,BnB 算法较 $RsS\text{-}Kconv$ 算法有较高的平均检查次数, 这样可以用额外的检查次数抵消需求不确定性的影响. 补货成本越高, 检查周期越长, 导致库存水平的不确定性增大, 因此相对最优性差距增大.

需求不确定性与相对最优差距之间有明显相关性: 随着 σ 增加, 相对最优差距也随之增加. 比较需求量的不同模式, 可以发现 $RsS\text{-}Kconv$ 算法在 (INC) 模式中表现最好, 相对最优性差距为 0 ; 在 (DEC) 模式中表现最差, 相对最优性差距最大. 平均而言, 相对最优性差距在 10 和 15 个周期数时分别为 1.32% 和 1.49% .

		相对最优性差距%	时间 (s)		检查次数		成本误差%	
			BnB	$RsS\text{-}Kconv$	BnB	$RsS\text{-}Kconv$	BnB	$RsS\text{-}Kconv$
K	20	0	354.19	31.74	8	8	0.2908	0.2908
	40	0.432	64.98	8.05	8	8	0.2935	0.3429
	80	0.9581	91.03	11.94	6	5	0.1139	0.0675
	160	1.711	93.47	11.41	6	4	0.0081	0.1807
	320	2.872	88.12	11.63	5	3	0.0976	0.0407
W	20	2.872	88.12	11.63	5	3	0.0976	0.0407
	40	1.3963	91.22	11.86	4	3	0.0969	0.0394
	80	0	99.54	12.35	3	3	0.037	0.037
	160	0	70.34	11.96	2	2	0.1978	0.1923
	320	0	51.42	12.07	2	2	0.1726	0.1678
σ	0.1	0	19.37	3.76	2	2	0.0208	0.0208
	0.2	0.7057	40.42	6.65	3	3	0.1087	0.0037
	0.3	1.8228	66.11	8.5	4	3	0.0704	0.0576
	0.4	2.872	88.12	11.63	5	3	0.0976	0.0407
模式	STA	2.0073	22.25	3.48	5	3	0.0715	0.0103
	INC	0	70.24	11.15	3	3	0.1514	0.1514
	DEC	2.872	88.12	11.63	5	3	0.0976	0.0407
	LCY1	1.9234	68.36	9.1	5	3	0.1029	0.1387
	LCY2	0.3416	77.27	13.22	4	3	0.0469	0.0698
	RAND	3.6185	128.12	15.07	6	4	0.0612	0.0158
平均		1.3202	85.02	26.98	4.55	3.55	0.1117	0.0975

表 1: $T = 10$:BnB 与 $RsS\text{-}Kconv$

在计算时间方面,BnB 算法的用时明显大于 $RsS\text{-}Kconv$ 算法的用时, 在 $T = 10$ 时, 两者的差距较小, 但会随着 T 的增大逐渐扩大. 原因是初始参数的变化对 BnB 的影响较大、对 $RsS\text{-}Kconv$ 算法的影响较小. 成本参数对 $RsS\text{-}Kconv$ 算法的性能几乎没有影响, 但对 BnB 算法的剪枝效果有影响. 例如 $T = 15$ 时, 低检查成本的实例比高检查成本的实例花费的时间多出将近 6 倍. 需求量的不确定性会影响这两种方法的性能, 然而 K-凸性的使用可以减少这种影响. 在 15 周期实例中, $\sigma = 0.4$ 时 $RsS\text{-}Kconv$ 算法的计算量比 $\sigma = 0.1$ 时增加了不到 1 倍, 而 BnB 算法的计算量增加了 5 倍以上.

5.3 实例分析

在本节中, 我们将分析两个实例, 以更好地理解计算的策略之间的差异. 第一个例子展示了一种 $RsS\text{-}Kconv$ 算法得到的策略是最优策略的情况, 第二个例子展示了一种 $RsS\text{-}Kconv$ 算法得到的策略不是最优策略的情况.

$RsS\text{-}Kconv$ 算法在计算解决方案时, 只考虑未来时期的预期需求, 忽略在时期 t 开始时的初始库存水平 I_{t-1} , 而 BnB 算法测试了搜索过程中所有可能的前几个时期的补货组合. 例如, 当出现 I_{t-1} 稍大于 s_t 但远小于 S_t 时,BnB 算法会根据此情况增加检查次数, 而 $RsS\text{-}Kconv$ 算法不受到 I_{t-1} 的影响. 对于需求量高不确定和递减模式 (DEC), I_{t-1} 的波动较大,BnB 算法会根据初始库存及时调整检查计划, 而 $RsS\text{-}Kconv$ 算法无法及时调整, 这种差异会恶化 $RsS\text{-}Kconv$ 算法的性能.

		相对最优性差距%	时间 (s)		检查次数		成本误差%	
			BnB	$RsS\text{-}Kconv$	BnB	$RsS\text{-}Kconv$	BnB	$RsS\text{-}Kconv$
K	20	0	1755.71	43.4	12	12	0.1904	0.1904
	40	0	2528.33	43.44	11	11	0.189	0.189
	80	0.6759	4038.23	43.69	10	8	0.1367	0.0344
	160	1.8275	4253.11	43.46	9	6	0.1337	0.2124
	320	3.3484	9969.25	43.77	7	4	0.2959	0.0763
W	20	3.3484	9969.25	43.77	7	4	0.2959	0.0763
	40	1.523	4574.1	43.58	6	4	0.108	0.0742
	80	0	4374.41	54.83	4	4	0.0703	0.0703
	160	0	2678.72	44.13	3	3	0.0827	0.0827
	320	0	1358.48	44.4	3	3	0.0722	0.0722
σ	0.1	0	824.85	23.81	3	3	0.0859	0.0859
	0.2	0.3551	2057.1	24.84	4	4	0.013	0.127
	0.3	2.2366	3682.06	34.12	6	4	0.0159	0.2088
	0.4	3.3484	9969.25	43.77	7	4	0.2959	0.0763
模式	STA	2.6284	2415.41	16.85	7	4	0.0923	0.0353
	INC	0.9725	2879.8	45.58	6	4	0.0914	0.0725
	DEC	3.3484	9969.25	43.77	7	4	0.2959	0.0763
	LCY1	1.5386	2609.75	30.82	7	5	0.0651	0.1133
	LCY2	1.5381	3288.36	43.24	7	5	0.2093	0.0448
	RAND	2.8863	6114.01	53.98	10	5	0.0027	0.1124
平均		1.4788	4465.4715	40.4625	6.8	5.05	0.1371	0.1015

表 2: $T = 15$:BnB 与 $RsS\text{-}Kconv$

观察表 1, 当参数为 $K = 20, W = 320, \sigma = 0.1$, 需求为 (INC) 模式时,$RsS\text{-}Kconv$ 算法相对最优性差距较小, 选取该组参数计算实例如下:

时期		1	2	3	4	5	6	7	8	9	10	理论成本
BnB	γ_t	1	0	0	1	0	0	0	0	0	0	1493
	S_t	264			248							
	s_t	208			220							
$RsS\text{-}Kconv$	γ_t	1	0	0	1	0	0	0	0	0	0	1493
	S_t	264			248							
	s_t	208			220							

表 3: $T = 10, K = 20, W = 320, \sigma = 0.1, pattern$ =INC

当参数为 $K=320, W=20, \sigma=0.4$, 需求为 (DEC) 模式时,$RsS\text{-}Kconv$ 算法相对最优性差距和非最优策略百分比较大, 选取该组参数计算实例如下:

时期		1	2	3	4	5	6	7	8	9	10	理论成本
BnB	γ_t	1	0	0	1	1	1	0	1	0	0	1793
	S_t	324			237	186	139		56			
	s_t	220			48	42	64		25			
$RsS\text{-}Kconv$	γ_t	1	0	0	1	0	0	0	1	0	0	1845
	S_t	295			243				56			
	s_t	211			174				25			

表 4: $T=10, K=320, W=20, \sigma=0.4, pattern$ =DEC

6 探究与拓展

$RsS\text{-}SDP\text{-}Kconv$ 法在求 R_t 的值时, 需要遍历库存值以找到最小的成本对应的库存. 我们猜想, 若在查找最小值时, 使用我们已学过的查找优化方法, 就可以很快的找到最小值, 以此迅速地得到 R 值. 但这样, 我们需要面临一个问题: 在计算 $E[C_{t+R_t}(I_t+Q_t-d_{t,t+R_t})]$ 时, 本质上还是需要遍历库存的值. 这跟我们想要优化的本质有冲突, 所以我们猜想, 松弛 $E[C_{t+R_t}(I_t+Q_t-d_{t,t+R_t})]$ 的计算, 或许会得到不错的结果, 于是, 我们列出如下的递归表达式:

$$\hat{C}_t(I_{t-1}) = min\{f_t(I_{t-1}, Q_t, R_t) + min(\hat{C}_{t+R_t}(I_t+Q_t-d_{t,t+R_t}))\}$$

边界条件为：$\hat{C}_{T+1}=0$

事实上, 如果从 T 开始向左递归, 那么 $min(\hat{C}_{t+R_t}(I_t+Q_t-d_{t,t+R_t}))$ 其实就是一个常数, 故只需要对 $f_t(I_{t-1}, Q_t, R_t)$ 求最小值即可. 类似上述, 可以将 $f_t(I_{t-1}, Q_t, R_t)$ 写成如下表达式：

$$f_t(x, y, R_t) = K1(y-x>0) + W + L_t(y)$$

其中,$L_t(y) \triangleq \sum_{i=1}^{R_t} E[hmax(y-d_{t,t+i}, 0) + bmax(d_{t,t+i}-y, 0)]$

在前文已经证明过 $L_t(y)$ 是 0-凸函数, 再引入以下命题：

命题. *若 $f(x)$ 是 0-凸函数, 那么 $f(x)$ 是下凸函数.*

证明. 由于 $f(x)$ 是 0-凸函数, 则对任意 $a, b>0$, 有：

$$f(x+a) - f(x) - a[\frac{f(x)-f(x-b)}{b}] \geq 0$$

取 $x_1=x-b$,$x_2=x+a$, 有 $x=(x_2-x_1)\frac{b}{a+b} = \frac{a}{a+b}x_1 + \frac{b}{a+b}x_2 \triangleq \lambda x_1 + (1-\lambda)x_2$, 代入即得：

$$f(x_2) - f(\lambda x_1 + (1-\lambda)x_2) - \frac{\lambda}{1-\lambda}[f(\lambda x_1 + (1-\lambda)x_2) - f(x_1)] \geq 0$$

这等价于

$$\lambda f(x_1) + (1-\lambda) f(x_2) \geq f(\lambda x_1 + (1-\lambda)x_2)$$

证毕. □

故有上述知 L_t 为 0-凸函数, 为了方便计算, 不妨设在求 R 的时候,S 和 s 相等, 即把原问题简化为 (R,S) 问题, 则 $f_t(x,y,R_t)$ 显然为 0-凸函数, 并且它还是连续的. 故若 f_t 有极小值点, 则所有极小值点对应的极小值都是相同的, 即只有一个最小值.

在求最小值的时候, 我们采用折半查找法进行查找, 由于在实际算法中, 我们所遍历点均为离散点, 故若相邻的点的函数值相同, 必有最小值点在这些点之间, 这个时候我们取较小值为上界, 就不会丢失最小值. 若相邻点的函数值不相等, 则根据函数值的比较, 我们可以更新上下界, 具体算法伪代码如下：

Algorithm 8: *RsS-Binary-Kconv*

```
for t = T : −1 : 1 do
    cost_best(t) = ∞
    r(t) = ∞
    for r = 1 : 1 : T − t + 1 do
        inf = inv_min
        sup = inv_max
        while inf + 1 < sup do
            S = (inf + sup)//2
            temp_1 = K + W + cost(t, s, r)
            temp_2 = K + W + cost(t, s + 1, r)
            if temp_1 == temp_2 then
                sup = S
                break
            end
            if temp_1 > temp_2 then
                inf = S
            else
                sup = S
            end
        end
        cost_now = K + W + cost(t, sup, r)
        if cost_now < cost_best(t) then
            cost_best(t) = cost_now
            r(t) = r
        end
    end
end
```

Algorithm 9: $cost(t,i,r)$

$cost_{temp} = 0$
if $r == 1$ **then**
  $temp+ = cost_{best}(t+1)$
else
  if $r! = 1$ **then**
    $temp+ = E[cost(t+1, i-d, r-1)]$
  end
  $temp+ = E[h\,max(i-d,0) + b\,max(d-i,0)]$
end
return $temp$

我们将 $RsS\text{-}Binary\text{-}Kconv$ 算法与 $RsS\text{-}Kconv$ 算法的计算时间进行比较, 设置参数与 5.1 节相同, 绘制图 2

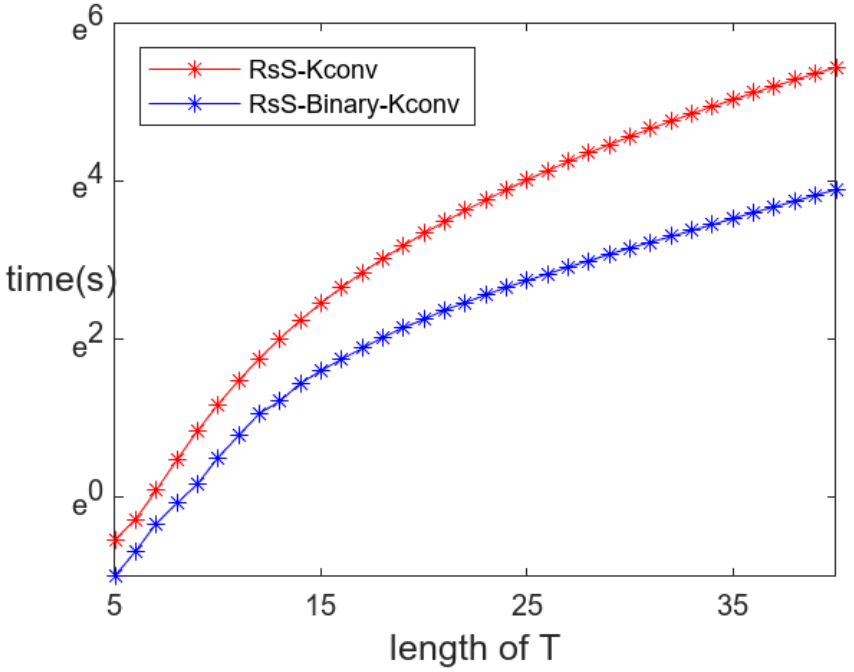

图 2: $T = 15$:$RsS\text{-}Binary\text{-}Kconv$ 与 $RsS\text{-}Kconv$

可以看出 $RsS\text{-}Binary\text{-}Kconv$ 在用时上相对于 $RsS\text{-}Kconv$ 得到了进一步的优化, 下表展示了 $RsS\text{-}Binary\text{-}Kconv$ 的相对最优性差距

		相对最优性差距%		时间 (s)		检查次数		成本误差%	
				Binary	*RsS-Kconv*	*Binary*	*RsS-Kconv*	*Binary*	*RsS-Kconv*
K	20	0	0	8.01	43.4	12	12	0.1904	0.1904
	40	0	0	9.37	43.44	11	11	0.189	0.189
	80	0.5535	0.6759	11.53	43.69	8	8	0.1285	0.0344
	160	2.9061	1.8275	13.56	43.46	5	6	0.0198	0.2124
	320	4.0356	3.3484	17.29	43.77	4	4	0.2429	0.0763
W	20	4.0356	3.3484	17.29	43.77	4	4	0.2429	0.0763
	40	2.1793	1.523	17.78	43.58	4	4	0.2363	0.0742
	80	0.578	0	18.5	54.83	3	4	0.0891	0.0703
	160	0	0	20.09	44.13	3	3	0.0827	0.0827
	320	0	0	21.47	44.4	3	3	0.0722	0.0722
σ	0.1	0	0	10.86	23.81	3	3	0.0859	0.0859
	0.2	0.3551	0.3551	9.99	24.84	4	4	0.127	0.127
	0.3	2.5646	2.2366	13.48	34.12	4	4	0.1787	0.2088
	0.4	4.0356	3.3484	17.29	43.77	4	4	0.2429	0.0763
模式	STA	2.6284	2.6284	6.45	16.85	4	4	0.0353	0.0353
	INC	0.9725	0.9725	19.95	45.58	4	4	0.0725	0.0725
	DEC	4.0356	3.3484	17.29	43.77	4	4	0.2429	0.0763
	LCY1	2.3909	1.5386	10.16	30.82	4	5	0.1281	0.1133
	LCY2	2.1718	1.5381	14.63	43.24	4	5	0.134	0.0448
	RAND	2.8863	2.8863	19.62	53.98	5	5	0.1124	0.1124
平均		1.8164	1.4788	14.7305	40.4625	4.85	5.05	0.1427	0.1015

表 5: $T = 15$:*RsS-Kconv* 与 *RsS-Binary-Kconv*

时期		1	2	3	4	5	6	7	8	9	10	理论成本
RsS-Binary-Kconv	γ_t	1	0	1	0	0	1	0	0	0	0	1899
	S_t	228		231			145					
	s_t	139		155			105					
RsS-Kconv	γ_t	1	0	0	1	0	0	0	1	0	0	1845
	S_t	295			243				56			
	s_t	211			174				25			

表 6: $T = 10, K = 320, W = 20, \sigma = 0.4, pattern$ =DEC

时期		1	2	3	4	5	6	7	8	9	10	理论成本
RsS-Binary-Kconv	γ_t	1	0	0	1	0	0	0	0	0	0	1493
	S_t	264			248							
	s_t	208			220							
RsS-Kconv	γ_t	1	0	0	1	0	0	0	0	0	0	1493
	S_t	264			248							
	s_t	208			220							

表 7: $T = 10, K = 20, W = 320, \sigma = 0.1, pattern$ =INC

7 结论

本文主要研究了 (s, S) 问题与 (R, s, S) 问题, 介绍了 *BnB* 算法、*RsS-SDP* 算法和 *RsS-SDP-Kconv* 算法, 并提出了对 *RsS-SDP-Kconv* 算法的改进算法 *RsS-Binary-Kconv*. 传

统算法中, 通过动态规划方程, 将所有变量集中至目标函数后一一遍历求出最优策略. 然而, 传统方法计算量巨大从而导致无法在实际问题中使用.BnB 算法通过剪枝可以在一定程度上减少遍历次数从而提高效率, 但剪枝率和具体参数相关, 在多数情况下也难以在实际问题中使用.$RsS\text{-}SDP$ 算法使用了启发法的思想, 通过引入松弛条件, 用局部近似最优 R_t 代替全局最优, 在舍弃部分精度的条件下, 可以提高计算效率. 更进一步, 利用 **Scarf** 提出的 K-凸性, 对 $RsS\text{-}SDP$ 算法进行简化得到 $RsS\text{-}SDP\text{-}Kconv$ 算法. 在此算法的基础上, 我们尝试在对 I_t 的遍历时使用折半查找代替逐个遍历, 得到 $RsS\text{-}Binary\text{-}Kconv$ 算法.

在数值实验中, 我们先比较了 BnB 算法、$RsS\text{-}SDP$ 算法和 $RsS\text{-}SDP\text{-}Kconv$ 算法的算法效率, 发现 $RsS\text{-}SDP$ 算法效率最差,$RsS\text{-}SDP\text{-}Kconv$ 算法在绝大多数情况下优于 BnB 算法, 在周期数增加时, 两者差距明显增大. 之后我们比较了 BnB 算法和 $RsS\text{-}SDP\text{-}Kconv$ 算法在不同参数下的表现, 结果显示两者仅在极端情况下差距明显, 但在大多数情况下表现接近. 在计算时间方面, 对于任意参数组合,$RsS\text{-}SDP\text{-}Kconv$ 算法都显著优于 BnB 算法. 另外我们还比较了新提出的 $RsS\text{-}Binary\text{-}Kconv$ 算法与 $RsS\text{-}SDP\text{-}Kconv$ 算法的表现, 发现两个算法在精度表现上接近, 但 $RsS\text{-}Binary\text{-}Kconv$ 算法可以显著减少计算时间.

参考文献

[1] Scarf, Herbert. 1959. “The optimality of (s, S) policies in the dynamic inventory problem.”

[2] Visentin, Andrea, Steven Prestwich, Roberto Rossi, and S Armagan Tarim. 2021. “Computing optimal (R, s, S) policy parameters by a hybrid of branch-and-bound and stochastic dynamic programming.” European Journal of Operational Research .

[3] nd S Armagan Tarim. 2012. “Stat Í ic-dynamic uncertainty strategy for a single-item stochastic inventory control problem.” Omega 40 (3): 348–357.

附录

这里是实验过程中使用的 python 代码

随即库存策略类函数

```
import numpy as np
from scipy.stats import poisson
from scipy.stats import norm
import random
import math

class InventoryInstance:
    def __init__(self):
        # Number of periods in the time horizon 最大时期数
        self.n = 1
        # Costs
        # Holding cost per unit per period 滞销惩罚
        self.ch = 0
        # Penalty cost per unit per period 缺货惩罚
        self.cp = 0
        # Linear production/ordering cost per unit 线性的订购成本
        self.cl = 0
        # Fixed ordering cost 固定的订购成本
        self.co = 0
        # Fixed review cost 固定的审查成本
        self.cr = 0

        # Service level 服务水平
        self.alpha = 0

        # Initial inventory 初始库存
        self.init_inv = 0
        # Maximum value of the demand with non null probability
            我认为是单个时期最多累积的需求量
        self.max_demand = 60
        # Demand probability distribution 需求的密度函数（离散形式）
        self.prob = []
        # Maxtrix containing the convolutions of the demand probability
            distribution 需求概率分布卷积
        self.conv_prob = []
        # Average values of the demand distribution 需求分布的均值
        self.means = []
        # Coefficient of variation in case of a normal distributed deman 分布系数
        self.cv = 1 / 3.0
        # Type of demand distribution - poisson or normal 需求分布型
        self.dem_type = "poisson"
```

```
    # Maximum and minimum inventory level 最大库存量和最小库存量,设置为需求的n倍
    self.max_inv_level = self.n * self.max_demand
    self.min_inv_level = -self.n * self.max_demand
    # Review times, if fixed 想要固定的审查时间
    self.rev_time = []

def max_inv_bouding(self, threshold): # 设定库存量上下界,取需求的某个区间
    可以是负数 并且确定了需求的分布
    demand = [0 for _ in range(self.n * self.max_demand + 1)] #
        n个时期,最多可以累积n * max_demand的需求
    for i in range(self.max_demand + 1):
        demand[i] = self.prob[0][i]
    for i in range(1, self.n):
        temp = [0 for _ in range(self.n * self.max_demand + 1)]
        for a in range((self.n - 1) * self.max_demand + 1):
            for b in range(self.max_demand + 1):
                temp[a + b] += self.prob[i][b] * demand[a] # 求出需求为a+b的概率
        demand = temp
    j = self.n * self.max_demand
    pdf_dem = demand[j]
    while pdf_dem < threshold: # 从后往前 将发生概率比较小的需求视为不可能事件
        j -= 1
        pdf_dem += demand[j - 1]

    self.max_inv_level = j # 将库存的最大最小值设为可能发生的需求量
    self.min_inv_level = -j//10 + self.init_inv

def probability_convolution(self): # 需求卷积分布函数
    self.conv_prob = np.zeros((self.n, self.n, self.n * self.max_demand + 1)) #
        创造三维数组

    for i in range(self.n):
        for j in range(self.max_demand+1):
            self.conv_prob[i][i][j] = self.prob[i][j]
        # self.conv_prob[i][i] = self.conv_prob[i][i] / sum(self.conv_prob[i][i])

    for i in range(0, self.n):
        for i2 in range(i+1, self.n):
            for j in range((i2-i)*self.max_demand +1):
                for j2 in range(self.max_demand+1):
                    self.conv_prob[i][i2][j+j2] += self.conv_prob[i][i2-1][j] *
                        self.prob[i2][j2]
            self.conv_prob[i][i2] = self.conv_prob[i][i2] /
                sum(self.conv_prob[i][i2])

# Probability distribution generators based on the means
    基于均值的概率分布生成器（意思是下面这几个包括了不一样的分布类型的需求）
def gen_poisson_probability(self, mu): # 泊松分布的概率分布
    self.prob = [[0.0 for _ in range(0, self.max_demand +1)] for _ in
        range(self.n)] # n行max_demand+1列
```

```
    for i in range(self.n):
        for j in range(0, self.max_demand +1):
            self.prob[i][j] = poisson.pmf(j, mu) # 需求为j的概率（泊松分布）
                赋值到prob里

def gen_fix_probability(self, avg): # 均值分布的概率分布
    self.prob = [[0.0 for _ in range(0, self.max_demand +1)] for _ in
        range(self.n)]
    if not (2 <= avg <= self.max_demand - 2):
        print("average outside interval allowed")
        return
    for i in range(self.n):
        self.prob[i][avg-2] = 0.1
        self.prob[i][avg-1] = 0.2
        self.prob[i][avg] = 0.4
        self.prob[i][avg+1] = 0.2
        self.prob[i][avg+2] = 0.1

def gen_bin_probability(self): # 是对二元分布类型需求的概率分布
    if self.max_demand != 1:
        print("Error - In binary demand max demand has to be equal to 1")
    self.prob = [[0.5, 0.5]for _ in range(self.n)]

def gen_non_stationary_normal_demand(self, threshold): #
    多个均值的正态分布需求类型的概率分布
    self.dem_type = "normal"
    if len(self.means) != self.n:
        print("wrong size self.means vector")
        return
    self.max_demand = int(norm(loc = max(self.means), scale =
        self.cv*max(self.means)).ppf(1-threshold)) #
        这个函数好像是求正态分布的累积分布函数的逆函数的
    self.prob = [[0.0 for _ in range(0, self.max_demand +1)] for _ in
        range(self.n)]
    for i in range(self.n):
        norm_dist = norm(loc=self.means[i], scale = self.cv*self.means[i])
        for j in range(self.max_demand +1):
            self.prob[i][j] = norm_dist.pdf(j) # x=j的概率密度
        self.prob[i] = self.prob[i]/sum(self.prob[i]) #扩充为概率和为1
    self.max_inv_level = self.n * self.max_demand
    self.min_inv_level = - self.n * self.max_demand + self.init_inv

def gen_non_stationary_poisson_demand(self, sigma,threshold): ## fix this VISE
    多个均值的泊松分布的概率分布
    if len(self.means) != self.n:
        print("wrong size self.means vector")
        return
    meanstemp=[0 for _ in range(self.n)]
    for i in range(self.n):
        temp= np.random.normal(self.means[i], sigma, 1)
```

```
            temp=temp.tolist()
            meanstemp[i]=temp[0]
        self.means=meanstemp
        print(self.means)
        self.max_demand = int(poisson.ppf(loc = 0 , mu = max(self.means), q =
            (1-threshold)))
        self.prob = [[0.0 for _ in range(0, self.max_demand +1)] for _ in
            range(self.n)]
        for i in range(self.n):
            for j in range(self.max_demand +1):
                if self.means[i] != 0 :
                    self.prob[i][j] = poisson.pmf(k = j, mu = self.means[i])
                else:
                    self.prob[i][j] = poisson.pmf(k = j, mu = 0.1) #
                        这些都和上面的过程很像
            self.prob[i] = self.prob[i]/sum(self.prob[i])
        #self.max_inv_level = self.n * self.max_demand
        #self.min_inv_level = - self.n * self.max_demand + self.init_inv

    # Pattern generator, they are generlly found in literature 模式生成器
        means数组的生成过程
    def gen_means(self, type):
        self.means = [0 for _ in range(self.n)]
        if type == "LCYA":
            for i in range(self.n):
                self.means[i] = round(19 * math.exp(-(i-12)**2) / 5)
        elif type == "SIN1":
            for i in range(self.n):
                self.means[i] = round(70 * math.sin(0.8*(i+1)) +80)
        # Form Rossi et al. 2011
        elif type == "P1": # Form Rossi et al. 2011
            for i in range(self.n):
                self.means[i] = round(50 * (1 + math.sin(math.pi * i /6)))
        elif type == "P2": # Form Rossi et al. 2011
            for i in range(self.n):
                self.means[i] = round(50 * (1 + math.sin(math.pi * i /6))) + i
        elif type == "P3": # Form Rossi et al. 2011
            for i in range(self.n):
                self.means[i] = round(50*(1 + math.sin(math.pi * i /6))) + self.n - i
        elif type == "P4": # Form Rossi et al. 2011
            for i in range(self.n):
                self.means[i] = round(50 * (1 + math.sin(math.pi * i /6))) + min(i,
                    self.n-i)
        elif type == "P5": # Form Rossi et al. 2011
            for i in range(self.n):
                self.means[i] = random.randint(0,100)
            # NEW PATTERNS FOR THE PAPER
        elif type == "STA": # new patterns for the paper
            for i in range(self.n):
                self.means[i] = 50
```

```
    elif type == "INC":
        for i in range(self.n):
            self.means[i] = round((2 * i + 1) * 100 / (2 * self.n))
    elif type == "DEC":
        for i in range(self.n):
            self.means[i] = round(100 - (2 * i + 1) * 100 / (2 * self.n))
    elif type == "LCY1": #the lifecycles can be simplified
        c = 0
        for i in range(int(self.n / 3)):
            self.means[i] = round((2 * i + 1) * 75 / (2 * int(self.n/3)))
        for i in range(int(self.n / 3), 2 * int(self.n / 3) + self.n % 3):
            self.means[i] = 75
        for i in range(2 * int(self.n / 3) + self.n % 3, self.n):
            self.means[i] = round(75 - (2 * c + 1) * 75 / (2 * int(self.n/3 +
                0.5)))
            c += 1
    elif type == "LCY2":
        c = 0
        for i in range(int(self.n / 2)):
            self.means[i] = round((2 * i + 1) * 100 / (2 * int(self.n/2)))
        for i in range(int(self.n / 2), self.n):
            self.means[i] = round(100 - (2 * c + 1) * 100 / (2 * int(self.n/2
                +0.5)))
            c += 1
    elif type == "ERR":
        for i in range(self.n):
            self.means[i] = random.randint(50, 100)
    elif type == "SIN":
        for i in range(self.n):
            self.means[i] = round(50 * (1 + math.sin(math.pi * i / 6)))
    return self.means

# It generates a single value for the demand in period t according to the pdf
    of d_t
def gen_demand(self, t): #
    t为第t个时期,求最小的需求i,使得需求小于等于i的概率大于等于某个随机概率val
    val = random.random() # 生成[0.0,1.0)范围内的随机浮点数
    pdf = 0.0
    for i in range(0, self.max_demand+1):
        pdf = pdf + self.prob[t][i]
        if pdf >= val:
            return i
    return self.max_demand
```

sS-SDP 初始求解器

```
'''
sS
Stochastic Dynamic Programming solution to (s, S)-
'''
from util import policy as pol

class sS_SDP:
    # Common attributes of all the solver
    name = "sS_SDP"
    id = 101

    def __init__(self):
        # Text to be printed after the solving
        self.message = ""

        # Attributes specific to the solver
        self.__opt_cost = []
        self.__opt_q = []
        self.expected_cost = 0

    def solve(self, inst):
        n = inst.n
        max_demand = inst.max_demand
        ch = inst.ch
        cp = inst.cp
        co = inst.co
        cr = inst.cr
        prob = inst.prob
        max_inv_level = inst.max_inv_level
        min_inv_level = inst.min_inv_level
        self.__opt_cost = [[float("inf") for _ in range(min_inv_level,
            max_inv_level + 1)] for _ in range(n + 1)]
        self.__opt_q = [[max_inv_level for _ in range(min_inv_level, max_inv_level
            + 1)] for _ in range(n + 1)]

        for i in range(min_inv_level, max_inv_level +1):
            self.__opt_cost[n][i] = 0

        for t in range(n - 1, -1, -1):
            for i in range(min_inv_level, max_inv_level +1):
                bsf_cost = float("inf")
                bsf_q = float("inf")
                q_limit = max_inv_level - i
                for q in range(0, q_limit + 1):
                    temp = 0 if q == 0 else co
                    for d in range(0, max_demand+1):
                        close_inv = i + q - d
                        if close_inv >= 0:
```

```
                    temp = temp + prob[t][d] * (ch * close_inv +
                        self.__opt_cost[t + 1][close_inv])
                if min_inv_level < close_inv < 0:
                    temp = temp + prob[t][d] * (-cp * close_inv +
                        self.__opt_cost[t + 1][close_inv])
                if close_inv <= min_inv_level:
                    temp = temp + prob[t][d] * (-cp * close_inv +
                        self.__opt_cost[t + 1][min_inv_level])
            if temp < bsf_cost:
                bsf_cost = temp
                bsf_q = q

        self.__opt_cost[t][i] = bsf_cost + cr
        self.__opt_q[t][i] = bsf_q
s = [-1] * n
S = [-1] * n
for t in range(n):
    for i in range(0, max_inv_level +1):
        if self.__opt_q[t][i] > 0:
            s[t] = i
            S[t] = i + self.__opt_q[t][i]

self.expected_cost = self.__opt_cost[0][0]

policy = pol.InventoryPolicy()
policy.name = self.name
policy.order_quantity_type = "dynamic"
policy.n = n
policy.s = s
policy.S = S
policy.R = [1] * n
policy.expected_cost = self.expected_cost

return policy
```

BnB-SDP 求解器

```
'''
RsS
Binary tree solution with Branch and Bound and Dynamic Programming Bounds with
    randomised tree search
It contains some prints useful for the toy problem printing
'''
import random
from util import policy as pol

class RsS_BranchAndBound:


    def __init__(self):
        # Text to be printed after the solving
        self.message = ""

        # Attributes specific to the solver
        self.__opt_cost = []
        self.expected_cost = 0
        self.msf = float("inf")
        self.__best_s = []
        self.__best_ss = []
        self.threshold = 0.001
        self.prune_count = 0
        self.__count = 1
        self._use_random = False
        self._use_heuristic_sp = False
        self._use_heuristic_sdp = False
        self.name = "RsS_BandB"
        self.id = 304

    @property
    def use_random(self):
        return self._use_random

    @use_random.setter
    def use_random(self, use_random):
        if use_random:
            self._use_random = True
            self.name = "RsS_BandBrand"
            self.id = 305

    def solve(self, inst):
        n = inst.n
        max_demand = inst.max_demand
        ch = inst.ch
        cp = inst.cp
        co = inst.co
```

```
cr = inst.cr
prob = inst.prob
init_inv = inst.init_inv
max_inv_level = inst.max_inv_level
min_inv_level = inst.min_inv_level
orig_p = 0
self.msf = float("inf")
orig_msf = self.msf
self.__count = 1

self.__opt_cost = [[0.0 for _ in range(min_inv_level, max_inv_level + 1)]
    for _ in range(n + 1)]
rev = [0 for _ in range(n)]
s = [0 for _ in range(n)]
ss = [0 for _ in range(n)]
min_dem = [0 for _ in range(n)]
max_dem = [max_demand for _ in range(n)]
for i in range(n):
    temp = prob[i][0]
    for j in range(1, max_demand + 1):
        if temp < self.threshold:
            temp += prob[i][j]
            min_dem[i] += 1
        else:
            break
    temp = prob[i][max_demand]
    for j in range(max_demand, -1,-1):
        if temp < self.threshold:
            temp += prob[i][j]
            max_dem[i] -= 1
        else:
            break
# print(min_dem)
# print(max_dem)

min_cost = [[0 for _ in range(min_inv_level, max_inv_level + 1)] for _ in
    range(n)]
for i in range(min_inv_level, max_inv_level + 1):
    if i > init_inv - min_dem[0]:
        min_cost[1][i] = cr + co + ch * max(0,i) - cp * min(0, i)
    else:
        if i >= 0:
            min_cost[1][i] = ch * i
        else:
            min_cost[1][i] = - cp * i
for t in range(2, n):
    temp_min = float('inf')
    for i in range(min_inv_level, max_inv_level + 1):
        if min_cost[t - 1][i] < temp_min:
            temp_min = min_cost[t - 1][i]
```

```
            min_cost[t][i] = temp_min
        for i in range(min_inv_level, max_inv_level + 1 - max_dem[t]):
            if i > 0:
                # I can speed it up by moving a min outside this cycle and
                    computing it once only, assigning it to
                # the min cost and avoid a linear operation here
                min_cost[t][i] = min(cr + co + i * ch + min_cost[t][i],
                                     i * ch + min([min_cost[t-1][x] for x in
                                         range(i + min_dem[t-1], i +
                                         max_dem[t-1])]))
            else:
                min_cost[t][i] = - cp * i + min([min_cost[t-1][x] for x in
                    range(i + min_dem[t-1], i + max_dem[t-1]+1)])
                # min_cost[t][i] = - cp * i + min(
                #     [min_cost[t - 1][x] for x in range(i + min_dem[t - 1], i +
                    max_dem[t - 1])])
        for i in range(max_inv_level + 1 - max_dem[t], max_inv_level + 1):
            min_cost[t][i] = cr + co + min_cost[t][i]


    def cost(t, i):
        temp = cr
        for d in range(max_demand + 1):
            if i - d >= max_inv_level:
                temp = temp + (ch * (i - d) + self.__opt_cost[t +
                    1][max_inv_level]) * prob[t][d]
            if i - d <= min_inv_level:
                temp = temp - (cp * (i - d) - self.__opt_cost[t +
                    1][min_inv_level]) * prob[t][d]
            if 0 < i - d < max_inv_level:
                temp = temp + (ch * (i - d) + self.__opt_cost[t + 1][i - d]) *
                    prob[t][d]
            if min_inv_level < i - d <= 0:
                temp = temp - (cp * (i - d) - self.__opt_cost[t + 1][i - d]) *
                    prob[t][d]
        return temp

    def preorder_traversal(t, r):
        self.__count += 1
        rev[t] = r
        if t == 0:
            if r == 1:
                i = max_inv_level
                tmp1 = cost(t, i)
                tmp2 = cost(t, i - 1)
                if tmp2 > tmp1:
                    print("Error increase max_demand")
                msf = float("inf")
                while True:
                    i = i - 1
```

```
                tmp2 = cost(t, i)
                if tmp2 < tmp1 and tmp2 <= msf:
                    msf = tmp2
                    ss[t] = i
                tmp1 = tmp2
                # if i <= min_inv_level:
                #     print("Error 1")
                if tmp2 > msf + co:
                    break
            s[t] = i
            tmp1 = cost(t, ss[t]) + co
            if init_inv <= s[t]:
                self.__opt_cost[t][init_inv] = tmp1
            else:
                self.__opt_cost[t][init_inv] = cost(t, init_inv)
        else:
            temp = 0
            for d in range(0, max_demand+1):
                close_inv = init_inv - d
                if close_inv >= 0:
                    temp += prob[t][d] * (ch * close_inv + self.__opt_cost[t
                        + 1][close_inv])
                if min_inv_level < close_inv < 0:
                    temp += prob[t][d] * (-cp * close_inv + self.__opt_cost[t
                        + 1][close_inv])
                if close_inv <= min_inv_level:
                    temp += prob[t][d] * (-cp * close_inv + self.__opt_cost[t
                        + 1][min_inv_level])
            self.__opt_cost[t][init_inv] = temp

        if self.__opt_cost[0][init_inv] < self.msf:
            self.msf = self.__opt_cost[0][init_inv]
            self.__best_s = [s[i] if rev[i] == 1 else float('inf') for i in
                range(n)]
            self.__best_ss = [ss[i] if rev[i] == 1 else float('inf') for i in
                range(n)]
    else:
        if r == 1:
            i = max_inv_level
            tmp1 = cost(t, i)
            self.__opt_cost[t][i] = tmp1
            tmp2 = cost(t, i-1)
            if tmp2 > tmp1:
                print("Error increase max_demand")
            msf = float("inf")
            while True:
                i = i - 1
                tmp2 = cost(t, i)
                self.__opt_cost[t][i] =tmp2
                if tmp2 < tmp1 and tmp2 <= msf:
```

```
                    msf = tmp2
                    ss[t] = i
                tmp1 = tmp2
                if i <= min_inv_level:
                    print("Error 1")
                if tmp2 > msf + co:
                    break
            s[t] = i
            tmp1 = cost(t, ss[t]) + co
            for i in range(min_inv_level, s[t] +1):
                self.__opt_cost[t][i] = tmp1
        else:
            for i in range(min_inv_level, max_inv_level + 1):
                temp = 0
                for d in range(0, max_demand+1):
                    close_inv = i - d
                    if close_inv >= 0:
                        temp += prob[t][d] * (ch * close_inv +
                            self.__opt_cost[t + 1][close_inv])
                    if min_inv_level < close_inv < 0:
                        temp += prob[t][d] * (-cp * close_inv +
                            self.__opt_cost[t + 1][close_inv])
                    if close_inv <= min_inv_level:
                        temp += prob[t][d] * (-cp * close_inv +
                            self.__opt_cost[t + 1][min_inv_level])
                self.__opt_cost[t][i] = temp
        if min([x + y for x,y in zip(self.__opt_cost[t], min_cost[t])]) <
             self.msf:
            if self.use_random:
                if random.random() > 0.5:
                    preorder_traversal(t - 1, 1)
                    preorder_traversal(t - 1, 0)
                else:
                    preorder_traversal(t-1, 0)
                    preorder_traversal(t-1, 1)
            else:
                preorder_traversal(t - 1, 1)
                preorder_traversal(t - 1, 0)

preorder_traversal(n-1, 0)
preorder_traversal(n-1, 1)

self.prune_count = 2**(n+1) -1 - self.__count

if self.msf == orig_msf:
    self.message += "Solution was already optimal\n"
    orig_p.pruning_percentage = self.prune_count/(2**(n+1)-1)
    return orig_p

self.expected_cost = self.msf
```

```
        policy = pol.InventoryPolicy()
        policy.name = self.name
        policy.order_quantity_type = "dynamic"
        policy.n = n
        policy.s = self.__best_s
        policy.S = self.__best_ss
        policy.R = [0 if self.__best_s[i] == float('inf') else 1 for i in range(n)]
        policy.expected_cost = self.msf
        policy.pruning_percentage = self.prune_count/(2**(n+1)-1)

        return policy
```

```
'''
RsS
Stochastic Dynamic Programming solution with k-convexity
Memoized cost function
'''
from util import policy as pol


class RsS_SDP_KConv_Memo:
    # Common attributes of all the solver
    name = "RsS_SDP_KConv_Memo_v4"
    id = 310

    def __init__(self):
        # Text to be printed after the solving
        self.message = ""

        # Attributes specific to the solver
        self.expected_cost = 0
        self.rev_time = []
        self.__opt_cost = []
        self.__opt_r = []

    def solve(self, inst): ### VISE fix capacity issues
        n = inst.n
        ch = inst.ch
        cp = inst.cp
        co = inst.co
        cr = inst.cr
        max_inv_level = inst.max_inv_level
        min_inv_level = inst.min_inv_level
        prob = inst.prob
        max_demand = inst.max_demand
        self.__opt_cost = [[float("inf") for _ in range(min_inv_level,
            max_inv_level +1)] for _ in range(n+1)]
        self.__opt_r = [0 for _ in range(n+1)]
        s = [float("inf") for _ in range(n)]
        ss = [float("inf") for _ in range(n)]
        R = [0 for _ in range(n)]

        for i in range(min_inv_level, max_inv_level +1):
            self.__opt_cost[n][i] = 0

        min_inv_level = [min_inv_level for _ in range(n+1)]
        min_inv_level[0] = 0
        for i in range(n):
            min_inv_level[i+1] = max(min_inv_level[i+1], min_inv_level[i] -
                (len(prob[i])-1))
```

```
memo = dict()
def cost(t, i, r):
    if (t,i,r) in memo:
        return memo[(t,i,r)]
    temp = 0.0
    for d in range(len(prob[t])):
        if r == 1:
            if i-d <= min_inv_level[t+1]:
                temp += prob[t][d] * self.__opt_cost[t+1][min_inv_level[t+1]]
            else:
                temp += prob[t][d] * self.__opt_cost[t+1][i-d]
        else:
            temp += prob[t][d] * cost(t+1,i-d,r-1)

        if i-d >= 0:
            temp += prob[t][d] * ch * (i-d)
        else:
            temp += prob[t][d] * cp * (d-i)
    memo[(t,i,r)] = temp
    return temp

for t in range(n - 1, -1, -1):
    msf_all = float("inf")
    for r in range(1, n - t + 1):
        temp_S = float("inf")
        i = max_inv_level
        tmp1 = cr + cost(t, i, r)
        tmp2 = cr + cost(t, i-1, r)
        if tmp2 > tmp1:
            print("Error increase max_demand")
        msf = float("inf")
        while True:
            i = i - 1
            tmp2 = cr + cost(t, i, r)
            if tmp2 < tmp1 and tmp2 <= msf:
                msf = tmp2
                temp_S = i

            tmp1 = tmp2
            # if i <= min_inv_level[t]: Old Error 1, no need to keep it
            #     print("Error 1")
            if tmp2 > msf + co:
                break
        temp_s = i
        tmp1 = cr + cost(t, temp_S, r) + co
        if tmp1 < msf_all:
            msf_all = tmp1
            s[t] = temp_s
            ss[t] = temp_S
```

```
            self.__opt_r[t] = r

            for i in range(min_inv_level[t], max_inv_level +1):
                if i <= s[t]:
                    self.__opt_cost[t][i] = tmp1
                else:
                    self.__opt_cost[t][i] = cr + cost(t,i, r)
    i = 0
    t = 0
    for r in range(1, n-t+1):
        i = inst.init_inv
        tmp1 = cost(t, i, r)
        if tmp1 < self.__opt_cost[t][i]:
            s[t] = i-1
            ss[t] = i-1
            self.__opt_r[t] = r
            self.__opt_cost[t][i] = tmp1
            i = r

    self.expected_cost = self.__opt_cost[0][inst.init_inv]

    j=0
    while j < n:
        R[j] = 1
        j += self.__opt_r[j]

    for i in range(n):
        if R[i] == 0:
            s[i] = float("inf")
            ss[i] = float("inf")

    if ss[0] == -1:
        R[0] = 0
        s[0] = float("inf")
        ss[0] = float("inf")

    policy = pol.InventoryPolicy()
    policy.name = self.name
    policy.order_quantity_type = "dynamic"
    policy.n = n
    policy.s = s
    policy.S = ss
    policy.R = R
    policy.expected_cost = self.expected_cost

    return policy
```

RsS-Binary-Kconv 求解器第一部分

```
'''
+ memoization of the cost function
+ filter of the cycle length
+ binary search of min immediate cost
'''
from util import policy as pol


class RS_SDP_Binary:
    # Common attributes of all the solver
    name = "RS_SDP_Binary"
    id = 202

    def __init__(self):
        # Text to be printed after the solving
        self.message = ""

        # Attributes specific to the solver
        self.__opt_cost = []
        self.expected_cost = 0

    def solve(self, inst):
        n = inst.n
        ch = inst.ch
        cp = inst.cp
        co = inst.co
        cr = inst.cr
        max_inv_level = inst.max_inv_level
        min_inv_level = inst.min_inv_level
        prob = inst.prob
        self.__opt_cost = [float("inf") for _ in range(n + 1)]
        opt_S = [max_inv_level for _ in range(n+1)]
        opt_r = [0 for _ in range(n+1)]

        self.__opt_cost[n] = 0

        memo = dict()
        def cost(t, i, r):
            if (t,i,r) in memo:
                return memo[(t,i,r)]
            temp = 0.0

            if r == 1:
                temp += self.__opt_cost[t + 1]
            for d in range(len(prob[t])):
                if r != 1:
                    temp += prob[t][d] * cost(t+1,i-d,r-1)
                if i-d >= 0:
```

```
            temp += prob[t][d] * ch * (i-d)
        else:
            temp += prob[t][d] * cp * (d-i)
    memo[(t,i,r)] = temp
    return temp

for t in range(n - 1, -1, -1):
    bsf_cost = float("inf")
    bsf_S = float("inf")
    bsf_r = float("inf")

    for r in range(1, n - t + 1):
        s_inf = 0
        s_sup = max_inv_level-1
        while s_inf + 1 < s_sup:
            S = (s_sup + s_inf+1)//2
            temp = cr + co + cost(t,S,r)
            temp2 = cr + co + cost(t,S+1,r)
            i = 1
            while temp == temp2:
                i+=1
                print("it happends")
                temp2 = cr + co + cost(t, S + i, r)
            if temp > temp2:
                s_inf = S
            else:
                s_sup = S

        temp_cost = cr + co + cost(t, s_sup, r)

        if temp_cost < bsf_cost:
            bsf_cost = temp_cost
            bsf_S = s_sup
            bsf_r = r
        elif temp_cost - self.__opt_cost[t+r] > bsf_cost: # if the immediate
             cost is bigger than the bsf prune
            break

    self.__opt_cost[t] = bsf_cost
    opt_S[t] = bsf_S
    opt_r[t] = bsf_r
j = 0
if opt_S[0] <= inst.init_inv:
    self.__opt_cost[0] -= cr + co
    j = opt_r[0]

while j < n:
    R[j] = 1
    j += opt_r[j]
```

```
        policy = pol.InventoryPolicy()
        policy.name = self.name
        policy.order_quantity_type = "dynamic"
        policy.n = n
        policy.R = R

        return policy
```

RsS-Binary-Kconv 求解器第二部分

```
'''
sS
Stochastic Dynamic Programming solution with Kconv
'''
from util import policy as pol

class sS_SDPKConvexity:
    # Common attributes of all the solver
    name = "sS_SDP_KConv"
    id = 102

    def __init__(self):
        # Text to be printed after the solving
        self.message = ""

        # Attributes specific to the solver
        self.expected_cost = 0
        self.rev_time = []

    def solve(self, inst):
        n = inst.n
        max_demand = inst.max_demand
        ch = inst.ch
        cp = inst.cp
        co = inst.co
        cr = inst.cr

        if self.rev_time == []:
            rev_time = [1] * n
        else:
            rev_time = self.rev_time

        prob = inst.prob
        max_inv_level = inst.max_inv_level
        min_inv_level = inst.min_inv_level
        J_new = [float("inf") for _ in range(min_inv_level, max_inv_level + 1)]
        J_old = [0.0 for _ in range(min_inv_level, max_inv_level + 1)]
        s = [float("inf") for _ in range(n)]
        ss = [float("inf") for _ in range(n)]

        def cost(t, i):
            temp = 0.0
            for d in range(max_demand + 1):
                if i - d >= max_inv_level:
                    temp = temp + (ch * (i - d) + J_old[max_inv_level]) * prob[t][d]
                if i - d <= min_inv_level:
                    temp = temp - (cp * (i - d) - J_old[min_inv_level]) * prob[t][d]
                if 0 < i - d < max_inv_level:
```

```
            temp = temp + (ch * (i - d) + J_old[i - d]) * prob[t][d]
        if min_inv_level < i - d <= 0:
            temp = temp - (cp * (i - d) - J_old[i - d]) * prob[t][d]
    return temp

for t in range(n - 1, -1, -1):
    if rev_time[t] == 1:
        i = max_inv_level
        tmp1 = cost(t, i)
        J_new[i] = tmp1
        tmp2 = cost(t, i-1)
        if tmp2 > tmp1:
            print("Error increase max_demand")
        msf = float("inf")
        while True:
            i = i - 1
            tmp2 = cost(t, i)
            J_new[i] = tmp2
            if tmp2 < tmp1 and tmp2 <= msf:
                msf = tmp2
                ss[t] = i
            tmp1 = tmp2
            if i <= min_inv_level:
                print("Error 1")
            if tmp2 > msf + co:
                break
        s[t] = i
        tmp1 = cost(t, ss[t]) + co
        for i in range(min_inv_level, int(s[t] +1)):
            J_new[i] = tmp1

        for i in range(min_inv_level, max_inv_level + 1):
            J_old[i] = J_new[i] + cr
    else:
        J_new = [0 for _ in range(min_inv_level, max_inv_level + 1)]
        for i in range(min_inv_level, max_inv_level + 1):
            temp = 0
            for d in range(0, max_demand+1):
                close_inv = i - d
                if close_inv >= 0:
                    temp += prob[t][d] * (ch * close_inv + J_old[close_inv])
                if min_inv_level < close_inv < 0:
                    temp += prob[t][d] * (-cp * close_inv + J_old[close_inv])
                if close_inv <= min_inv_level:
                    temp += prob[t][d] * (-cp * close_inv +
                        J_old[min_inv_level])
            J_new[i] = temp
        for i in range(min_inv_level, max_inv_level + 1):
            J_old[i] = J_new[i]
self.expected_cost = J_old[inst.init_inv]
```

```
        policy = pol.InventoryPolicy()
        policy.name = self.name
        policy.order_quantity_type = "dynamic"
        policy.n = n
        policy.s = s
        policy.S = ss
        policy.R = rev_time
        policy.expected_cost = self.expected_cost

        return policy
```