实验三: 0-1 背包问题求解

PB20061372　朱云沁　June 17, 2023

1 实验说明

1.1 实验目的

利用不同算法设计策略来求解组合优化问题.

1.2 实验内容

假设有 n 个物品和一个背包, 每个物品重量为 $0 < w_i < 100$, $i = 1, 2, \ldots, n$, 价值为 $0 < v_i < 100$, $i = 1, 2, \ldots, n$, 背包最大容量为 c, 请问该如何选择物品才能使得装入背包中的物品总价值最大? 最大价值为多少? 请按照如下要求完成算法:

1. 请利用分治法来求解该问题, 给出最优解值以及求解时间;
2. 请利用动态规划算法来求解该问题, 给出得到最优解值以及求解时间;
3. 请利用贪心算法来求解该问题, 给出得到的最优解值以及求解时间;
4. 请利用回溯法来求解该问题, 给出得到的最优解值以及求解时间;
5. 请利用分支限界法来求解该问题, 给出得到的最优解值以及求解时间;
6. 请利用蒙特卡洛算法来求解该问题, 给出得到的最优解值以及求解时间;
7. 请利用深度强化学习算法来求解该问题, 给出最优解值以及求解时间;
8. 给定相同输入, 比较上述算法得到的最优解值和求解时间. 当 n 比较大的时候, 上述算法运算时间可能很长, 请在算法中增加终止条件以确保在有限时间内找到最优解的值.

2 问题描述及建模

0-1 背包问题 (0-1 Knapsack) 是一种 NP 完全的组合优化问题. 定义 x_i 为第 i 个物品是否放入背包中, 则其公式化描述如下:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} v_i x_i \\ \text{s.t.} \quad & \sum_{i=1}^{n} w_i x_i \le c \\ & x_i \in \{0,1\}, i = 1, 2, \ldots, n \end{aligned}$$

解空间 $X = \{(x_1, x_2, \ldots, x_n) \mid x_i \in \{0,1\}, i = 1, 2, \ldots, n\}$ 的大小为 2^n, 故用穷举法求解该问题的时间复杂度为 $O(2^n)$.

2.1 分治法 (Divide and Conquer)

定义子问题 $m(i,c)$ 为: 从第 i 个至第 n 个物品中选取若干物品放入容量为 c 的背包中, 使得总价值最大, 即

$$m(i,c) = \max\left\{\sum_{j=i}^{n} v_j x_j \mid \sum_{j=i}^{n} w_j x_j \le c, x_j \in \{0,1\}, j = i, i+1, \ldots, n\right\}$$

则原问题为 $m(1,c)$. 注意到 $m(1,c)$ 具有最优子结构性质: 令 $(x_1, x_2, \ldots, x_n)$ 为 $m(1,c)$ 的一个最优解, 则 $(x_2, x_3, \ldots, x_n)$ 是 $m(2, c - w_1x_1)$ 的最优解. 否则, 假设 $(y_2, y_3, \ldots, y_n)$ 是 $m(2, c - w_1x_1)$ 的最优解, 有 $\sum_{i=2}^{n} v_i y_i > \sum_{i=2}^{n} v_i x_i$ 且 $\sum_{i=2}^{n} w_i y_i \le c - w_1 x_1$, 可以推出 $v_1x_1 + \sum_{i=2}^{n} v_i y_i > v_1 x_1 + \sum_{i=2}^{n} v_i x_i$ 且 $w_1x_1 + \sum_{i=2}^{n} w_i y_i \le c$, 与 $(x_1, x_2, \ldots, x_n)$ 是 $m(1,c)$ 的最优解矛盾.

由此, 有递归式

$$m(i,c) = \begin{cases} \max\{m(i+1,c), v_i + m(i+1, c - w_i)\}, & \text{if } w_i \le c \\ m(i+1,c), & \text{otherwise} \end{cases} \qquad i = 1, 2, \ldots, n-1$$

边界条件为 $m(n,c) = v_n \mathbb{I}(w_n \le c)$.

当 c 与 w_i, $i = 1, 2, \ldots, n$ 均为整数时, 递归树的大小为 $O(nc)$, 故分治法求解该问题的时间复杂度为 $O(nc)$. 伪代码如下

```
Algorithm 1: Divide and Conquer
Input: $n$, $c$, $w[1..n]$, $v[1..n]$.
Output: $\textsc{Knapsack}(1,c)$.
function $\textsc{Knapsack}(i,c)$:
  if $i = n$ then
      return $v_n \mathbb{I}(w_n \le c)$
  if $w_i > c$ then
      return $\textsc{Knapsack}(i+1,c)$
  return $\max\{\textsc{Knapsack}(i+1,c), v_i + \textsc{Knapsack}(i+1, c-w_i)\}$
```

2.2 动态规划 (Dynamic Programming)

上述分治法可能会重复求解一些子问题. 为了避免这种情况, 可以使用带备忘录的递归算法 (也称带备忘录的动态规划), 以空间换取时间; 或使用自底向上的动态规划, 伪代码如下

```
Algorithm 2: Dynamic Programming
Input: $n$, $c$, $w[1..n]$, $v[1..n]$.
Output: $m[1,c]$.
  Initialize an 2D array $m[1..n, 0..c]$ with 0.
  for $j = w_n$ to $c$ do
      $m[n,j] \leftarrow v_n$
  for $i = n-1$ downto 1 do
      for $j = 0$ to $w_i - 1$ do
          $m[i,j] \leftarrow m[i+1,j]$
      for $j = w_i$ to $c$ do
          $m[i,j] \leftarrow \max\{m[i+1,j], v_i + m[i+1, j-w_i]\}$
  return $m[1,c]$
```

应当声明的是, 上述算法逐一求解子问题, 但并非所有子问题的解均被使用, 这意味着可能存在过多冗余计算. 此外, 该算法只适用于 c 与 w_i, $i = 1, 2, \ldots, n$ 均为整数的情形.

2.3 贪心算法 (Greedy Algorithm)

考虑 0-1 背包问题的特殊情况. 如果 $w_1 \le w_2 \le \cdots \le w_n$ 且 $v_1 \ge v_2 \ge \cdots \ge v_n$, 则问题具有贪心选择性质: 每次求解 $m(i,c)$, 只要 $w_i \le c$, 就选择 $x_i = 1$, 进而求解唯一子问题 $m(i+1, c-w_i)$; 否则, 选择 $x_i = 0$, 进而求解唯一子问题 $m(i+1, c)$. 换言之, 我们逐一选择单位重量价值 $\frac{v_i}{w_i}$ 最大的物品 i, 直至背包装满. 下证, 该贪心解最优.

令 $(x_i, x_2, \ldots, x_n)$ 为子问题 $m(i,c)$ 的一个最优解. (1) 若 $x_i = 1$, 由可行性, 必有 $w_i \le c$. 显然, 该解符合贪心策略. (2) 若 $x_i = 0$ 且 $w_i > c$, 该解同样符合贪心策略. (3) 若 $x_i = 0$ 且 $w_i \le c$, 令 $x_k = 1$ 为最优解中下标最小的非零分量, 则必有 $w_k = w_i$, $v_k = v_i$, 从而 $(1, x_2, \ldots, x_{k-1}, 0, x_{k+1}, \ldots x_n)$ 同样为最优解且符合贪心策略; 否则, 若 $w_k > w_i$, $v_k < v_i$, 易知 $(1, x_2, \ldots, x_{k-1}, 0, x_{k+1}, \ldots x_n)$ 仍为可行解且总价值更大, 与 $(x_i, x_2, \ldots, x_n)$ 为最优解矛盾. 综上所述, 贪心策略可以得到一个最优解.

对于一般情况, 首先将物品按单位重量价值排序, 使得 $\frac{v_1}{w_1} \le \frac{v_2}{w_2} \le \cdots \le \frac{v_n}{w_n}$. 若此时满足 $w_1 \le w_2 \le \cdots \le w_n$ 且 $v_1 \ge v_2 \ge \cdots \ge v_n$, 则贪心算法可以得到最优解. 伪代码如下

```
Algorithm 3: Greedy Algorithm
Input: $n$, $c$, $w[1..n]$, $v[1..n]$.
Output: $m$.
  Let $k_i$ be the sorted index of $\frac{v[i]}{w[i]}$ such that $\frac{v[k_1]}{w[k_1]} \le \frac{v[k_2]}{w[k_2]} \le \cdots \le \frac{v[k_n]}{w[k_n]}$.
  $m \leftarrow 0$
  for $i = 1$ to $n$ do
      if $w[k_i] \le c$ then
          $m \leftarrow m + v[k_i]$
          $c \leftarrow c - w[k_i]$
```

上述算法的时间复杂度取决于排序操作. 若不存在排列使得 $w_1 \le w_2 \le \cdots \le w_n$ 与 $v_1 \ge v_2 \ge \cdots \ge v_n$ 同时满足, 则单位重量价值贪心策略不一定能得到最优解. 同样, 重量贪心策略 (按 w_i 排序) 与价值贪心策略 (按 v_i 排序) 也不一定能得到最优解.

2.4 回溯法 (Backtracking)

在分治法中, 我们实际上以递归的方式对解空间树进行了深度优先搜索 (Depth-First Search). 然而, 每次调用函数 Knapsack(i,c), 我们均分配新的内存用于记录当前参数 i, c 与函数返回值 $m(i,c)$, 造成大量空间浪费.

事实上, 若将当前待选择物品 i 与背包剩余容量 c 视为问题的状态, 以全局变量的形式存储, 每次递归调用结束后, 我们仍能以 $O(1)$ 时间回溯至上一状态. 为了避免比较函数返回值, 我们选择不在回溯的过程中构造最优解, 而是自顶向下地记录搜索路径和累计价值 cv, 更新最优价值 m .

经过上述调整, 我们得到了基本的回溯算法. 可行性约束函数 (Constraint) 缩小了解空间树的规模. 经过观察, 我们发现可以用如下限界函数 (Bound) 进一步将解空间树剪枝: 若 $\sum_{j=1}^{i-1} v_j x_j + \sum_{j=i}^{n} v_j \le m$, 即当前累计价值与子问题的价值上界之和不超过当前最优解, 则不再搜索该子树.

我们用数组 $r[1..n]$ 记录预处理得到子问题的价值上界. 伪代码如下

```
Algorithm 4: Backtracking
Input: $n$, $c$, $w[1..n]$, $v[1..n]$.
Output: $m$.
  $i \leftarrow 1$; $m \leftarrow 0$; $cv \leftarrow 0$; $r[n+1] \leftarrow 0$
  for $j = n$ downto 1 do
      $r[j] \leftarrow r[j+1] + v[j]$
  $\textsc{Knapsack}()$

function $\textsc{Knapsack}()$:
  if $i > n$ then
      return
  if $cv + r[i] \le m$ then                                   // Bound
      return
  if $w[i] \le c$ then                                        // Constraint
      $c \leftarrow c - w[i]$; $cv \leftarrow cv + v[i]$
      $m \leftarrow \max\{m, cv\}$
      $i \leftarrow i + 1$
      $\textsc{Knapsack}()$
      $i \leftarrow i - 1$
      $c \leftarrow c + w[i]$; $cv \leftarrow cv - v[i]$
  $i \leftarrow i + 1$;
  $\textsc{Knapsack}()$
  $i \leftarrow i - 1$
  return
```

2.5 分支限界法 (Branch and Bound)

与回溯法不同, 分支限界法通常采用广度优先 (Breadth-First) 或最佳优先 (Best-First) 的方式搜索解空间树. 在 0-1 背包问题中, 可用如下启发式函数 (Heuristic) 定义节点的优先级:

$$b = \sum_{j=1}^{i-1} v_j x_j + m_{\text{greedy}}(i,c) \tag{1}$$

其中, c 为当前背包剩余容量, $m_{\text{greedy}}(i,c)$ 为将子问题放宽为小数背包问题后的最优价值, 可由贪心算法得到. 显然, $m_{\text{greedy}}(i,c)$ 是子问题最优价值的一个上界, 因此 (1) 式实际上给出了一个限界函数.

我们利用启发式函数 b 同时定义了分支策略与限界策略, 得到优先队列式分支限界算法. 伪代码如下

```
Algorithm 5: Branch and Bound
Input: $n$, $c$, $w[1..n]$, $v[1..n]$.
Output: $m$.
  Let $k_i$ be the sorted index of $\frac{v[i]}{w[i]}$ such that $\frac{v[k_1]}{w[k_1]} \le \frac{v[k_2]}{w[k_2]} \le \cdots \le \frac{v[k_n]}{w[k_n]}$.
  Let $q$ be a priority queue (max-heap) of node states in the form of $(b, i, c, cv)$.
  $m \leftarrow 0$;
  $b \leftarrow \textsc{Bound}(1, c, 0)$
  $\textsc{Enqueue}(q, (b, 1, c, 0))$
  loop
      $(b, i, c, cv) \leftarrow \textsc{Dequeue}(q)$
      if $i > n$ or $c = 0$ then
          break
      if $b \le m$ then
          continue
      $\textsc{Enqueue}(q, (\textsc{Bound}(i+1, c, cv), i+1, c, cv))$
      if $w[i] \le c$ then
          $m \leftarrow \max\{m, cv + v[i]\}$
          $\textsc{Enqueue}(q, (b, i+1, c - w[i], cv + v[i]))$

function $\textsc{Bound}(i, c, cv)$:
  for $j \leftarrow i$ to $n$ do
      if $w[k_j] \le c$ then
          $c \leftarrow c - w[k_j]$
          $cv \leftarrow cv + v[k_j]$
      else
          break
  if $j \le n$ then
      $cv \leftarrow cv + \frac{v[k_j]}{w[k_j]} \cdot c$
  return $cv$
```

受限于搜索顺序, 我们需要同时维护优先队列中所有节点的状态, 因此空间开销远大于回溯法. 上述算法搜索到一个可行解后即终止, 由于启发式函数的良好性质, 该解必定最优.

2.6 蒙特卡洛算法 (Monte Carlo Algorithm)

蒙特卡洛算法泛指一类随机算法, 其输出解以一定的概率接近于问题的真实解, 且随着算法重复次数的增加, 输出解的正确概率逐渐提高. 通常, 蒙特卡洛算法依赖于重复的随机抽样来近似问题的解.

为了使用蒙特卡洛算法解决 0-1 背包问题, 我们首先定义解空间上的概率分布, 使得最优解为该分布的众数, 从而通过随机抽样得到的可能性最大. 一个常见的选择是玻尔兹曼 (Boltzmann) 分布:

$$p(x) \propto \exp\left(-\frac{E(x)}{T}\right) \tag{2}$$

其中, $E(x) = -\sum_{i=1}^{n} v_i x_i$ 为状态 x 的势能 (Potential Energy), 由该状态下背包总价值取负得到 (若 x 不可行, 则定义 $E(x) = +\infty$); $T > 0$ 为温度 (Temperature), 可以预先选定或随着算法的迭代逐渐降低. 由 (2) 式可知, 若状态 x 越接近最优解, 则背包价值越大, 能量越小, 该状态被抽样的概率 $p(x)$ 越大, 符合我们对 $p(x)$ 的要求.

此外, 观察发现, 温度 T 越低, 能量差对应的概率差越大, 算法越倾向于抽样能量较小的状态; 相反, 当温度 T 趋近于正无穷时, 算法从所有解中均匀抽样. 因此, 超参数 T 控制了算法对探索与利用 (Exploration-Exploitation) 的权衡, 对求解速度与解的质量有重要影响.

由于无法得知从概率 $p(x)$ 到 x 的逆映射, 直接从 (2) 式中独立同分布地抽样是困难且低效的. 马尔可夫链蒙特卡洛 (Markov Chain Monte Carlo, MCMC) 算法提供了一种解决方案: 从某个初始状态 $x^{(0)}$ 出发, 每次抽样结果 $x^{(t)}$ 均依赖于上一次抽样结果 $x^{(t-1)}$, 从而构造得一个马尔可夫链 $\{x^{(t)}\}_{t=0}^{\infty}$. 通过设计抽样机制, 使得该马尔可夫链的稳态分布为 $p(x)$, 最终以状态出现的频率近似概率 $p(x)$.

Metropolis–Hastings 是一种常用的 MCMC 算法, 其抽样机制如下:

1. 从某一提议 (Proposal) 分布 $q(x'|x^{(t-1)})$ 中抽样得到 x';
2. 计算接受率 (Acceptance Ratio) $\alpha(x', x^{(t-1)}) = \min\left\{1, \frac{p(x')q(x^{(t-1)}|x')}{p(x^{(t-1)})q(x'|x^{(t-1)})}\right\}$;
3. 以概率 $\alpha(x', x^{(t-1)})$ 接受 x', 令 $x^{(t)} = x'$; 否则拒绝 x', 保留 $x^{(t-1)}$.

由以上步骤, 有状态转移概率 $p(x'|x^{(t-1)}) = q(x'|x^{(t-1)})\alpha(x', x^{(t-1)})$. 容易证明, 该抽样机制满足细致平衡 (Detailed Balance) 条件 $p(x)p(x'|x) = p(x')p(x|x')$, 因此马尔可夫链的稳态分布为 $p(x)$.

在 0-1 背包问题中, 我们选取提议分布 $q(x'|x)$ 为 x 的邻域中的均匀分布, 即以等概率将状态 x 中任意一位取反 (对应放入背包或拿出背包), 因而 $q(x'|x) = q(x|x') = \frac{1}{n}$. 此时, 接受率 $\alpha(x', x) = \min\left\{1, \frac{p(x')}{p(x)}\right\} = \min\left\{1, \exp\left(\frac{E(x) - E(x')}{T}\right)\right\}$. 可见, 当 x' 为不可行解时, 能量为正无穷, 算法必然拒绝 x', 从而保证解的可行性; 当 x' 为可行解, 且价值较 x 更大时, 能量差 $E(x') - E(x) < 0$, 算法必然接受 x', 从而实现价值函数的优化; 当 x' 为可行解, 且价值较 x 更小时, 算法以一定概率接受 x', 从而保留对解空间的探索.

为了确保在有限时间预算内得到较优解, 兼顾解的质量与求解速度, 我们按照一定的退火方案 (Annealing Schedule) 逐渐减小温度 T, 从而使得算法在前期更多地探索解空间, 在后期更多地利用已有信息. 考虑到实现的简单性, 此处使用线性退火方案 $T = T_{\max} - \frac{\texttt{elapsed}}{\texttt{budget}}(T_{\max} - T_{\min})$, 其中 $T_{\max}$ 与 $T_{\min}$ 分别为初始温度与最终温度, `elapsed` 为算法已用时间, `budget` 为算法总时间预算.

综上, 我们得到了模拟退火 (Simulated Annealing) 算法的完整流程, 伪代码如下

```
Algorithm 6: Monte Carlo (Simulated Annealing)
Input: $n$, $c$, $w[1..n]$, $v[1..n]$, $T_{\max}$, $T_{\min}$, annealing schedule.
Output: $m$.
  $T \leftarrow T_{\max}$;
  Initialize $x[1..n]$ as zero vector.
  while $T > T_{\min}$ do
      Uniformly sample $i \in \{1, 2, \ldots, n\}$.
      if $x[i] = 0$ and $\sum_{j=1}^{n} w[j]x[j] + x[i] > c$ then
          continue
      $\Delta E \leftarrow (2x[i] - 1)v[i]$
      if $\Delta E < 0$ then
          $x[i] \leftarrow 1 - x[i]$
          $m \leftarrow \max\{m, \sum_{i=1}^{n} v[i]x[i]\}$
      else
          Uniformly sample $u \in [0, 1]$.
          if $u < \exp(-\Delta E / T)$ then
              $x[i] \leftarrow 1 - x[i]$
      Update $T$ according to annealing schedule.
```

2.7 深度强化学习 (Deep Reinforcement Learning)

近年来, 深度强化学习在搜索 NP 难的组合优化问题的近似最优解方面取得了巨大的成功. 现有工作可大致分为两类: (1) 利用深度强化学习模型直接构造问题的解; (2) 利用深度强化学习模型改进传统的局部搜索算法. 此处, 我们实现一种基于 REINFORCE 算法的模型, 通过学习得到的策略, 直接构造物品序列. 该部分主要参考如下文献:

1. Kool, W., Hoof, H. van, & Welling, M. (2019). Attention, Learn to Solve Routing Problems! *ICLR 2019*.
2. Kwon, Y.-D., Choo, J., Kim, B., Yoon, I., Gwon, Y., & Min, S. (2020). POMO: Policy Optimization with Multiple Optima for Reinforcement Learning. *NeurIPS 2020*.

类似于贪心算法的排序过程, 我们将问题的解定义为物品下标的一个排列 $(k_1, k_2, \ldots, k_n)$, 其中 $k_i \in \{1, 2, \ldots, n\}, k_i \ne k_j, \forall i \ne j$, 只需依序将物品放入背包, 直至背包装满. 问题的状态定义为当前背包剩余容量 $s^{(i)} = c - \sum_{j=1}^{i-1} w_{k_j}$. 问题的求解过程类似于选择排序, 假定第 i 步时已选择了物品 $\{k_1, k_2, \ldots, k_{i-1}\}$, 则行动 $a^{(i)}$ 的目标是从 $\{1, 2, \ldots, n\} \setminus \{k_1, k_2, \ldots, k_{i-1}\}$ 中选择下一个物品 k_i. 可见, 该问题的最优策略依赖于过去所有行动, 即

$$\pi_\theta^{(i)} = \begin{cases} p_\theta(a^{(1)}|s^{(1)}) & \text{if } i = 1 \\ p_\theta(a^{(i)}|s^{(i)}, a^{(1:i-1)}) & \text{if } i = 2, 3, \ldots, n \end{cases}$$

其中, $\pi_\theta^{(i)}, i = 1, 2, \ldots, n$ 为策略函数, θ 为可学习参数. 最终的解为所有行动的轨迹 (Trajectory) $\tau = (a^{(1)}, a^{(2)}, \ldots, a^{(n)})$, 其概率为

$$p_\theta(\tau|s^{(1)}) = p_\theta(a^{(1)}|s^{(1)}) \prod_{i=2}^{n} p_\theta(a^{(i)}|s^{(i)}, a^{(1:i-1)})$$

我们的目标是最大化期望累积奖励 $J(\theta, s) = \mathbb{E}_{p_\theta(\tau|s)}[R(\tau)]$. REINFORCE 是一种基于策略梯度的算法, 该算法将 $J(\theta, s)$ 的梯度定义如下

$$\nabla_\theta J(\theta, s) = \mathbb{E}_{p_\theta(\tau|s)}[(R(\tau) - b(s))\nabla_\theta \log p_\theta(\tau|s)]$$

其中, $b(s)$ 为基线函数 (Baseline), $A(\tau, s) = R(\tau) - b(s)$ 称作优势函数 (Advantage).

在 Policy Optimization with Multiple Optima (POMO, Kwon et al. 2020) 中, 模型每次从 N 个不同初始行动 $a_1^{(1)}, a_2^{(1)}, \ldots, a_N^{(1)}$ 出发, 同时推断出 N 个轨迹 $(\tau_1, \tau_2, \ldots, \tau_N)$, 从而将基线函数与策略梯度分别近似为

$$b(s) \approx \frac{1}{N}\sum_{i=1}^{N} R(\tau_i)$$

$$\nabla_\theta J(\theta, s) \approx \frac{1}{N}\sum_{i=1}^{N}(R(\tau_i) - b(s))\nabla_\theta \log p_\theta(\tau_i|s)$$

遵循 Kool et al. 2019 的思路, 受 Transformer 启发, 我们将策略网络设计为如下两部分: (1) Encoder, 利用多头注意力机制将物品 i 的特征向量 (w_i, v_i) 嵌入到隐空间; (2) Decoder, 将物品嵌入作为键和值, 上下文嵌入 (此处, 直接使用背包剩余容量 c) 作为查询, 利用注意力机制计算单步行动的概率分布. 具体的模型架构详见附件内的 TensorFlow 实现.

伪代码如下

```
Algorithm 7: Deep Reinforcement Learning (POMO)
function $\textsc{Training}$(training set $D$, number of initial states $N$, number of epochs $T$, batch size $B$, learning rate $\alpha$):
  Initialize policy network parameters $\theta$.
  for $i \leftarrow 1$ to $T$ do
      Sample a batch of $B$ problems $\{P_1, P_2, \ldots, P_B\}$ from $D$.
      Encode item data into embeddings for each problem in the batch.
      Select $N$ initial actions $\{(a_{i,1}^{(0)}, a_{i,2}^{(0)}, \ldots, a_{i,N}^{(0)})\}_{i=1}^{B}$ for each problem $P_i$ in the batch.
      Decode the policy and sample rollouts $\{(\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,N})\}_{i=1}^{B}$ from $N$ initial actions.
      Compute baselines $b_i \leftarrow \frac{1}{N}\sum_{j=1}^{N} R(\tau_{i,j})$, $i = 1, 2, \ldots, B$.
      Compute gradients $\nabla_\theta J(\theta, s) \leftarrow \frac{1}{BN}\sum_{i=1}^{B}\sum_{j=1}^{N}(R(\tau_{i,j}) - b_i)\nabla_\theta \log p_\theta(\tau_{i,j}|s)$.
      Update policy network parameters $\theta \leftarrow \theta + \alpha\nabla_\theta J(\theta, s)$.
  return $\theta$

function $\textsc{Inference}(n, c, w[1..n], v[1..n])$:
  Encode item data $(w[1..n], v[1..n])$ into embeddings.
  Select $N$ initial actions $\{(a_{i,1}^{(0)}, a_{i,2}^{(0)}, \ldots, a_{i,N}^{(0)})\}_{i=1}^{B}$.
  Decode the policy and select the best actions to form $N$ solutions $(\tau_{i,1}, \tau_{i,2}, \ldots, \tau_{i,N})$.
  return $\tau^* = \arg\max_{\tau_{i,j}} R(\tau_{i,j})$.
```

3 经典算法性能比较

我们使用 C++ 分别实现 0-1 背包问题的分治法, 动态规划算法, 贪心算法, 回溯法, 分支限界法和蒙特卡洛算法, 编译为静态链接库, 并在 Python 中调用.

为了反映算法性能随问题规模的变化, 我们选取 $n \in [5, 200]$ 进行多组实验. 考虑到背包容量 c 和物品数量 n 的比值可能会影响问题的求解难度, 我们固定 $c = 25n$. 对于每组 n 和 c, 我们均匀地抽取 $(0, 100)$ 范围内的整数作为物品的价值 $v[1..n]$ 和重量 $w[1..n]$, 将相同的问题输入到不同的算法中, 并统计每个算法的运行时间和优化间隙 (Optimality Gap). 其中, 优化间隙定义为算法得到的解与动态规划得到的最优解之间的相对误差.

为了避免分治法与回溯法花费的时间过长, 我们限制时间预算为 200 毫秒. 当程序运行时间超过该阈值时, 我们终止递归调用, 直接返回当前搜索到的价值最大的可行解. 出于公平性考虑, 我们将模拟退火算法的时间预算同样设置为 200 毫秒. 此外, 模拟退火的最大, 最小温度分别设置为 $T_{\max} = 1, T_{\max} = 0$.

为了减少不确定性, 对于每组 n 和 c, 我们以多线程的方式同时求解 100 个随机生成的问题, 取平均运行时间和平均优化间隙作为最终结果, 所得数据如表 1, 2 所示. 以 n 为横坐标, 将平均运行时间和平均优化间隙绘制成折线图, 如图 1 所示.

Table 1. Average time cost (milliseconds) of classical methods on different problem sizes.

n	D. & C.	DP	Greedy	Backtrack	B. & B.	MC
5	2.159	0.085	0.076	1.541	0.155	200.68
10	75.908	0.183	0.057	12.98	0.267	201.028
15	202.363	0.383	0.063	102.67	0.357	201.255
20	202.476	0.643	0.068	195.286	0.652	200.901
25	203.335	1.058	0.061	201.536	0.676	201.343
50	204.123	3.034	0.101	203.433	1.813	200.61
75	205.534	6.698	0.122	205.475	2.585	204.134
100	207.224	13.414	0.167	207.81	5.613	200.893
125	209.654	19.785	0.111	209.079	3.727	201.395
150	209.13	26.237	0.13	209.94	5.564	200.312
175	211.036	41.244	0.25	210.731	5.142	200.576
200	211.427	46.212	0.212	212.941	5.709	200.429

Table 2. Average optimality gap of classical methods on different problem sizes.

n	D. & C.	DP	Greedy	Backtrack	B. & B.	MC
5	0.0	0.0	0.023	0.0	0.0	0.0
10	0.0	0.0	0.017	0.0	0.0	0.0
15	0.121	0.0	0.009	0.004	0.0	0.003
20	0.192	0.0	0.007	0.07	0.0	0.011
25	0.242	0.0	0.003	0.139	0.0	0.014
50	0.313	0.0	0.003	0.254	0.0	0.019
75	0.348	0.0	0.001	0.298	0.0	0.025
100	0.367	0.0	0.001	0.32	0.0	0.026
125	0.364	0.0	0.001	0.339	0.0	0.029
150	0.363	0.0	0.001	0.344	0.0	0.029
175	0.373	0.0	0.001	0.349	0.0	0.03
200	0.376	0.0	0.0	0.348	0.0	0.035

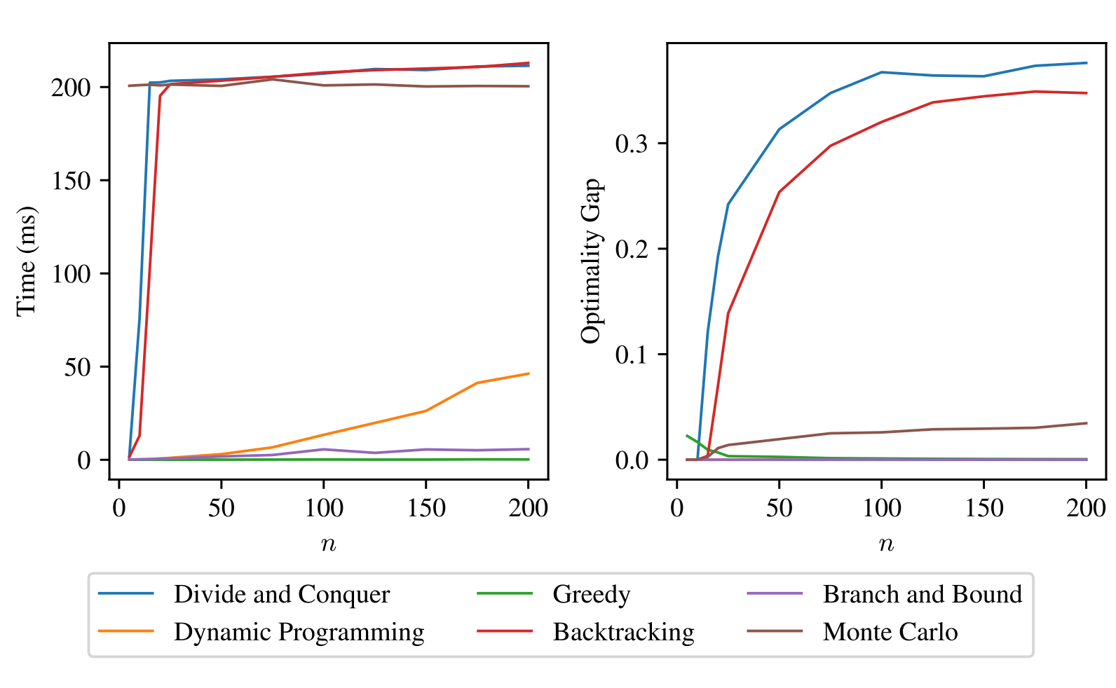

Figure 1. Time cost and optimality gap of classical methods on different problem sizes.

由以上图表, 作如下观察:

1. 当问题规模较小时, 分治法与回溯法均能正确求得最优解, 但其运行时间随着问题规模的增大指数增加, 以至在 $n = 15$ 时就已经超出预算, 提前终止运行, 返回的解与最优解相差较大. 这说明必须寻求其他方法, 用于改进暴力搜索的 $O(2^n)$ 时间复杂度.
2. 由于 c 与 w 均为整数, 动态规划能够在 $O(nc)$ 时间复杂度内求得真实的最优解, 其运行时间 (以及内存开销) 呈现二次增长的趋势. 当问题规模更大时, 该方法可能仍会超出预算.
3. 按单位重量价值排序的贪心算法总是能在最短时间内求得质量较高的解, 但并不保证解的最优性. 随问题规模的增大, 贪心算法求得正确解的概率越来越大, 优化间隙越来越小以至趋近于零. 从直觉上讲, 这是由于本实验中各物品重量范围固定, 随物品数量的增长, 使用不同重量组合填满背包的难度减小, 一定程度上缓解了贪心策略短视性的缺点.
4. 由于启发式函数的良好性质, 分支限界法能够在较短时间内求得真实的最优解, 显示出该方法的优越性. 然而, 应当注意, 对于个别难度较大或规模较大的问题, 分支限界法仍有可能超出时间 (及空间) 预算.
5. 模拟退火作为一种随机优化算法, 能够保证在时间预算内求得较好的近似解. 实验中, 该算法的优化间隙随问题规模的增大而增大, 意味着可能需要更多的迭代次数或超参数调优.

4 深度强化学习算法性能评估

我们使用 TensorFlow 完整实现 Policy Optimization with Multiple Optima 算法. 网络超参数如下: 物品嵌入维数为 32, 注意力机制的键的维数为 16, 注意力头数为 4, 前馈隐藏层维数为 64, 编码器层数为 2, 解码器中对数几率的裁剪范围为 $[-10, 10]$. 我们在 GTX 1080 Ti GPU 上训练 200 个 epoch. 其中, 每个 epoch 包含 5 个 batch, 每个 batch 包含 64 个问题实例, 每个问题均从 $n = 50$ 个不同初始物品分别出发得到 50 个行动轨迹, 并据此计算梯度更新参数. 为了提升模型的泛化能力, 我们使用随机浮点数作为 c, w, v, 并适当限制了 w 的下限, 放宽了 w 的上限. 我们使用 Adam 优化器, 采用固定的学习率 8×10^{-5} 以及固定的权值衰减 1×10^{-7}. 训练过程花费约 7 分钟. 损失函数变化如图 2 所示. 注意下图中, 训练集上的损失函数由随机抽样的行动轨迹计算得到, 而验证集上的损失函数由概率最大的行动轨迹计算得到.

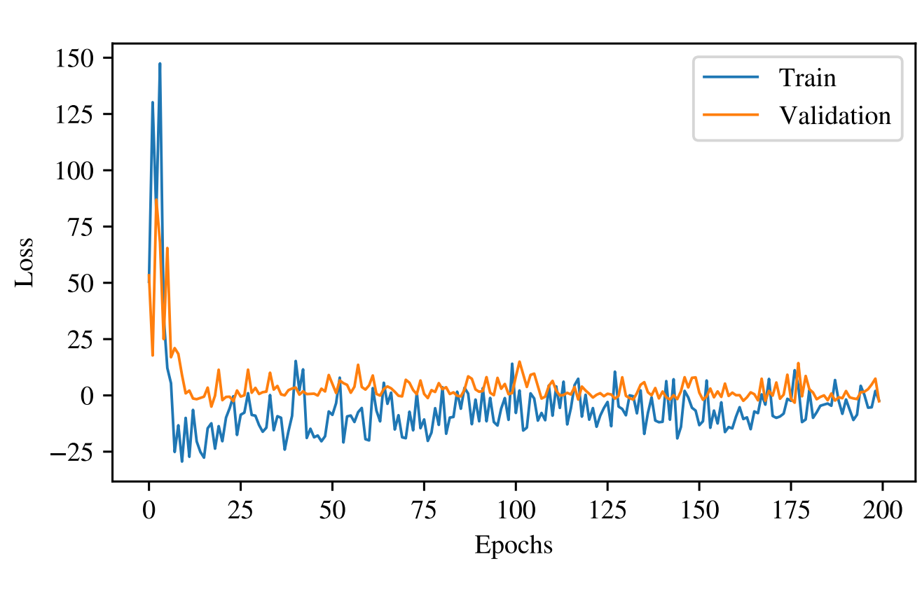

Figure 2. Learning curve of the REINFORCE agent.

我们将模型的性能与按单位重量价值排序的贪心策略以及模拟退火算法进行比较. 每次测试, 固定 $n = 50$, $c = 750$, 每个方法均使用相同输入, 模拟退火的时间预算限制为 200 毫秒. 应当声明的是, 我们的模型原生地支持如下特性: 从 N 个不同初始物品出发得到 N 个行动轨迹, 选取价值最大的解作为最终结果. 然而, 该行为可能会导致性能评估的不公平性. 因此, 我们作出限制, 只允许模型推断 1 个行动轨迹. 实验结果如表 3 所示.

Table 3. Performance comparison between deep reinforcement learning method and classical baselines.

Method	Time (ms)	Optimality Gap
Greedy Strategy	0.1011	0.003
Simulated Annealing	201.3374	0.0252
Policy Optimization w/ Multiple Optima	**166.8119**	**0.0022**

可见, 我们的深度强化学习模型超越了贪心策略的性能, 并拥有较短的推理时间, 展示出其优越性. 由于该模型架构的内在特征, 我们可以轻松地将其应用于任意问题规模 n 和 c, 并调整推理的超参数 N 达到最优性和效率的平衡. 由于时间有限, 此处不作过多测试.

5 结论

本实验中, 我们实现了求解 $0-1$ 背包问题的分治法, 动态规划算法, 贪心算法, 回溯法, 分支限界法和蒙特卡洛算法. 实验结果表明, 对于不同的问题规模和应用需求, 这些算法各有优劣. 我们还实现了一个基于注意力机制的深度强化学习模型, 并将其应用于 $0-1$ 背包问题. 该模型能够在较短时间内求得较好的近似解, 并且具有较好的泛化能力. 该模型的性能超越了贪心策略, 并且满足时间预算的要求.