wedge 積の高速化手法について

計数工学科 3 年 03-220602 浜口広樹

2023 年 2 月 5 日

1 問題設定

a を Q^{2p} サイズのテンソル、b を Q^{2q} サイズのテンソルとし、$N = p + q$ と定義する。

$$(a \wedge b)^{i_1,\cdots,i_p,i_{p+1},\cdots,i_{p+q}}_{j_1,\cdots,j_p,j_{p+1},\cdots,j_{p+q}} = \left(\frac{1}{N!}\right)^2 \sum_{\pi,\sigma} \epsilon(\pi)\epsilon(\sigma) a^{\sigma(i_1),\sigma(i_2),\cdots,\sigma(i_p)}_{\pi(j_1),\pi(j_2),\cdots,\pi(j_p)} b^{\sigma(i_{p+1}),\sigma(i_{p+2}),\cdots,\sigma(i_{p+q})}_{\pi(j_{p+1}),\pi(j_{p+2}),\cdots,\pi(j_{p+q})}$$

で定義される wedge 積について、その全要素を求めよ。
また、この時の計算量はいくらほどか。

2 前提

これを定義通り計算すると、まず、$a \wedge b$ が $Q^{2p+2q} = Q^{2N}$ 要素あって、各要素が $N!^2$ 個の置換からなるので、計算量は $\Theta\left(Q^{2N} N!^2\right)$ です。(ここで、Θ とは、$\mathcal{O}$ 記法よりもちょっと評価を厳密にした計算量評価を表す記号と思ってもらって差し支えありません)

これが、同値類などということを考えると、$\Theta\left({}_QC_N{}^2 \left({}_NP_{N/2}{}^2 + N!^2\right)\right)$ になり、そして最終的に $\Theta\left({}_QC_N{}^2 \left({}_NP_{N/2}{}^2 + N!^2\right)\right)$ まで落ちる、というのがこのメモの主張です。

一般的な $\mathcal{O}$ 記法で書くならば、${}_NP_p{}^2 << N!^2$ であることから、上記二つの計算量は共に $\mathcal{O}\left({}_QC_N{}^2 N!^2\right)$ と書くべきであり、定数倍 (約 1/2 倍) の差でしかありません。

ただ、(手柄を誇張するようなつもりはない上、そもそもそんなに難しい話ではないのですが、) 個人的にはやや意味のある定数倍かとは思っています。逆行列の計算などで、この 1/2 が付くことに意味があるとする文脈もあるからです。

また、topM を列挙するという際には明確な計算量削減になります。$N!^2$ の項が消えるので、${}_NP_p{}^2 << N!^2$ という差が生きてくるからです。

3　反対称性から分かること

考察をします。

まず、wedge 積の反対称性から、添字に (上側毎、下側毎で) 重複がある場合はゼロになります。また、上側毎、下側毎での順番を考慮しない添字集合の直積を考えると、これらは再び反対称性から ± 1 倍で計算出来ます。

これを同値類と見なすと、同値類の数が ${}_QC_N{}^2$ 個、同値類の代表元 (上側毎、下側毎で添字がソートされているような要素を指す) を計算するのに、

$$\sum_{\pi,\sigma} \epsilon(\pi)\epsilon(\sigma) a^{\sigma(i_1),\sigma(i_2),\cdots,\sigma(i_p)}_{\pi(j_1),\pi(j_2),\cdots,\pi(j_p)} b^{\sigma(i_{p+1}),\sigma(i_{p+2}),\cdots,\sigma(i_{p+q})}_{\pi(j_{p+1}),\pi(j_{p+2}),\cdots,\pi(j_{p+q})}$$

が必要なので、$n!^2$ 回の演算、そして、同じ同値類に属する要素を代表元から計算するのが、それぞれ $\mathcal{O}(1)$ で出来るので、また、同一の同値類に属する要素は合計で $N!^2$ 個あるので、合わせて、

$$\Theta\left({}_QC_N{}^2\left(N!^2 + N!^2\right)\right)$$

になります。

ここまでは割と自明な話かと思います。

(尤も、最初に見た時はこの時点でかなり驚きましたね……。たったの反対称性だけで計算量下限 (非零要素数) がオーダーの観点で達成できるとは思っていませんでしたので。なので、反対称性って偉いんだなという話を、趣味の記事に書こうと思っていました。

例えば行列式の計算も似たような事を考えると

$$\sum_{\sigma\in Aut(n)} \left(\mathrm{sgn}(\sigma)\prod_{i=1}^{n} a_{i,\sigma i}\right)$$

は一見 $\mathcal{O}((N!)N)$ の計算量が掛かりますが、三角行列などへ変換することで $\mathcal{O}\left(N^3\right)$ で計算出来ることは有名事実です。

これは、そもそも行基本変形などが反対称性に由来していると捉えられるので、根底に似た話があります。(私自身、最初は行列式がなぜ高速に計算できるのかをもう一度ちゃんと考察し直すところからスタートしました)

個人的にこの話は面白いなと感じている所です。)

4　代表元計算の高速化

ここで、先程代表元の計算の際に、

$$\sum_{\pi,\sigma}\epsilon(\pi)\epsilon(\sigma)a^{\sigma(i_1),\sigma(i_2),\cdots,\sigma(i_p)}_{\pi(j_1),\pi(j_2),\cdots,\pi(j_p)}b^{\sigma(i_{p+1}),\sigma(i_{p+2}),\cdots,\sigma(i_{p+q})}_{\pi(j_{p+1}),\pi(j_{p+2}),\cdots,\pi(j_{p+q})}$$

を計算するとしましたが、これは二重の $\sum$ になっているので無駄がありそうです。例えば、$\sum_\pi\sum_\sigma\cdots$ から、$(\sum_\pi\cdots)(\sum_\sigma\cdots)$ や、$\sum_\pi\cdots$ の形に出来れば嬉しさがあります。前者に関しては、恐らく厳しいはずですが、実は、後者は (本質的には) 似たようなことが可能です。

まず、今の状況設定をもう一度確認すると、$i_1,\cdots,i_{p+q},j_1,\cdots,j_{p+q}$ は固定されていて、かつ、i 毎、j 毎に関して、重複がありません。

この時、$\sigma(i_1),\sigma(i_2),\cdots,\sigma(i_p)$ が確定すると、b 側に関して $\sigma(i_{p+1}),\sigma(i_{p+2}),\cdots,\sigma(i_{p+q})$ は、$\{i_k|0\leq k\leq p+q\}\setminus\{\sigma(i_k)|0\leq k\leq p\}$ という集合の置換全体で表現されることが分かります。

なので、この b 側に関してあり得る置換全体というのを前計算しておけば $\sigma(i_1),\sigma(i_2),\cdots,\sigma(i_p)$ と $\pi(j_1),\pi(j_2),\cdots,\pi(j_p)$ の二つさえ確定させれば、残りは $\mathcal{O}(1)$ で計算できます。

さらに、$p\leq\frac{N}{2}$ が仮定出来ます。

$p>q$ の時、これの左右を入れ替えることを考えたいですが、これが、操作をしても何も変わらないということを以下に示します。

$$\begin{aligned}
&(a\wedge b)^{i_1,\cdots,i_p,i_{p+1},\cdots,i_{p+q}}_{j_1,\cdots,j_p,j_{p+1},\cdots,j_{p+q}}\\
&=\left(\frac{1}{N!}\right)^2\sum_{\pi,\sigma}\epsilon(\pi)\epsilon(\sigma)a^{\sigma(i_1),\cdots,\sigma(i_p)}_{\pi(j_1),\cdots,\pi(j_p)}b^{\sigma(i_{p+1}),\cdots,\sigma(i_{p+q})}_{\pi(j_{p+1}),\cdots,\pi(j_{p+q})}\\
&=\left(\frac{1}{N!}\right)^2\sum_{\pi,\sigma}\epsilon(\pi)\epsilon(\sigma)b^{\sigma(i_{p+1}),\cdots,\sigma(i_{p+q})}_{\pi(j_{p+1}),\cdots,\pi(j_{p+q})}a^{\sigma(i_1),\cdots,\sigma(i_p)}_{\pi(j_1),\cdots,\pi(j_p)}\\
&=\left(\frac{1}{N!}\right)^2\sum_{\pi,\sigma}\epsilon(\pi)\epsilon(\sigma)\epsilon(\tau)^2b^{\sigma(i_{\tau(p+1)}),\cdots,\sigma(i_{\tau(p+q)})}_{\pi(j_{\tau(p+1)}),\cdots,\pi(j_{\tau(p+q)})}a^{\sigma(i_{\tau(1)}),\cdots,\sigma(i_{\tau(p)})}_{\pi(j_{\tau(1)}),\cdots,\pi(j_{\tau(p)})}\\
&=\left(\frac{1}{N!}\right)^2\sum_{\pi,\sigma}\epsilon(\pi)\epsilon(\sigma)b^{\sigma(i_1),\cdots,\sigma(i_q)}_{\pi(j_1),\cdots,\pi(j_q)}a^{\sigma(i_{q+1}),\cdots,\sigma(i_{q+p})}_{\pi(j_{q+1}),\cdots,\pi(j_{q+p})}\\
&=(b\wedge a)^{i_1,\cdots,i_q,i_{q+1},\cdots,i_{q+p}}_{j_1,\cdots,j_q,j_{q+1},\cdots,j_{q+p}}\\
&=(b\wedge a)^{i_1,\cdots,i_p,i_{p+1},\cdots,i_{p+q}}_{j_1,\cdots,j_p,j_{p+1},\cdots,j_{p+q}}
\end{aligned}$$

ここで、4 行目から 5 行目に関して、$\epsilon(\tau)^2$ というが、$\pm1^2=1$ かつ、$0^2=0$ であることから、これは消すことが出来ます。また、τ とは添字のソートを表す置換です。

よって、$p\leq q$ として一般性を失わないので、$p\leq\frac{N}{2}$ となります。

以上を総括して、計算量は

$$\Theta\left({}_QC_N{}^2\left({}_NP_{N/2}{}^2+N!^2\right)\right)$$

になりました。
ただし、b についての前計算がまだあって、${}_QC_k{}^2$ 通りの組合せに対して、$q!^2$ 回の計算があれば、

$$\sum_{\pi,\sigma} \epsilon(\pi)\epsilon(\sigma) b^{\sigma(i_{p+1}),\sigma(i_{p+2}),\cdots,\sigma(i_{p+q})}_{\pi(j_{p+1}),\pi(j_{p+2}),\cdots,\pi(j_{p+q})}$$

が求まるので、正確には $+{}_QC_k{}^2 q!^2$ となって、

$$\Theta\left({}_QC_k{}^2 q!^2 + {}_QC_N{}^2\left({}_NP_p{}^2 + N!^2\right)\right) \quad (p < q)$$

と書くべきでしょう。
尤も、これは wedge 積を考える範囲では影響がありません。

5 実装

以上を基に実装したのが以下です。
かなり多くの定数倍高速化の工夫が組み込まれています。が、それは競プロの話なので、特筆すべきことはありません。

ソースコード 1 code

```
def fast_wedge(left_tensor: np.ndarray,
    right_tensor: np.ndarray,
    left_index_ranks: Tuple[int, int],
    right_index_ranks: Tuple[int, int],
    verbose: bool = True) -> np.ndarray:
    # 要請1:
       left_tensorの次元は、left_index_ranksの総和と一致する(
       openfermionにもある仕様)
    assert left_tensor.ndim == sum(left_index_ranks)
    # 要請2:
       right_tensorの次元は、right_index_ranksの総和と一致する(
       openfermionにもある仕様)
    assert right_tensor.ndim == sum(right_index_ranks)
    # 要請3: left_tensorとright_tensorでn_qubitsが同一である
    assert len(set(left_tensor.shape) | set(right_tensor.shape))
       == 1
    # 要請4: left_tensor_ranksは同じ数字のペアでなければならない(
       openfermionにはない仕様)
    assert left_index_ranks[0] == left_index_ranks[1]
    # 要請5: right_tensor_ranksは同じ数字のペアでなければならない
       (openfermionにはない仕様)
    assert right_index_ranks[0] == right_index_ranks[1]
    # 要請6: n_qubits >= p + q でなければならない(そうでなければ
       全て0)
```

```
17      assert left_tensor.shape[0] >= left_index_ranks[0]+
            right_index_ranks[0]
18
19      if left_index_ranks[0] > right_index_ranks[1]:
20      left_tensor, right_tensor =\
21      right_tensor, left_tensor
22      left_index_ranks, right_index_ranks =\
23      right_index_ranks, left_index_ranks
24
25      # 定数定義
26      p = left_index_ranks[0]
27      q = right_index_ranks[0]
28      N = p+q
29      N_fact_2 = math.factorial(N)**2
30      Q = left_tensor.shape[0]
31      idx_up = Q**N
32
33      # ランダムアクセスが必要、かつ、多次元配列のままだと遅いので
            、通常の一次元listを使用
34      tensor = [0.0+0.0j for _ in range(Q**(2*N))]
35      left_tensor_list = left_tensor.flatten().tolist()
36      right_tensor_list = right_tensor.flatten().tolist()
37
38      # 符号や順列などについての事前計算
39      fixed_N = _generate_fixed_parity_permutations(N)
40      fixed_q = _generate_fixed_parity_permutations(q)
41      fixed_Np = _generate_fixed_partial_perms(N, p)
42
43      # right_tensorについての事前計算
44      fixed_right_list = [0.0+0.0j for _ in range(Q**(2*q))]
45      for iq in combinations(range(Q), q):
46      for jq in combinations(range(Q), q):
47      right = 0.0+0.0j
48      for niq, parity1 in _generate_parity_permutations(iq, fixed_q
            ):
49          for njq, parity2 in _generate_parity_permutations(jq,
                fixed_q):
50              right += right_tensor_list[_getIdx(Q, *niq, *njq)] *
                    \
51                  parity1*parity2
52      fixed_right_list[_getIdx(Q, *iq, *jq)] = right
53
54      # 添字順序の逆転による影響を考慮する符号
55      sign_adjustment = (-1)**(p*q)
56
57      # 添え字についてソートされているものを代表元としてループを回
            す
```

```
    for ipiq, jpjq in tqdm(product(combinations(range(Q), p+q),
                                   combinations(range(Q), p+q)),
                           total=(math.factorial(Q)
                                  // math.factorial(p+q)
                                  // math.factorial(Q-(p+q)))**2,
                           disable=not verbose):
    parity_ipiq = _generate_parity_permutations(ipiq, fixed_N)
    parity_jpjq = _generate_parity_permutations(jpjq, fixed_N)
    ans = 0.0+0.0j

    # 代表元に当たる要素の計算
    for nip, niq, i_parity in _partial_perms(ipiq, fixed_Np):
    for njp, njq, j_parity in _partial_perms(jpjq, fixed_Np):
        ans += left_tensor_list[_getIdx(Q, *nip, *njp)] * \
            fixed_right_list[_getIdx(
                Q, *niq, *njq)] * i_parity*j_parity
    ans /= N_fact_2
    ans *= sign_adjustment

    # 同値類に属する要素へ、代表元の値を利用して計算
    for nipiq, i_parity in parity_ipiq:
    nipiq_idx = _getIdx(Q, *nipiq)*idx_up
    for njpjq, j_parity in parity_jpjq:
        tensor[nipiq_idx+_getIdx(Q, *njpjq)] = ans*i_parity*
            j_parity

    return np.array(tensor).reshape(tuple(Q for _ in range(2*N)))
```

6 結論

以上が今回の高速化手法です。

恐らく、まだアルゴリズム改善の余地があっても全くおかしくは無いのですが、私は分かっていません。