4.1. Suffix Array

本節では、本論文で多用している Suffix Array という検索用データ構造について解説する。

Suffix Array とは高速な文字列検索を可能にするデータ構造である。UNIX の grep コマンドのような「テキストに対するあらゆる部分文字列の検索」を高速[1] に行なうことができる。ただし、あらかじめ検索用インデックス (= Suffix Array) を作成しておく必要がある。

Suffix Array の特徴を簡単にまとめておく。

- どんな部分文字列でも検索可能。日本語テキストへのインデクシングで、形態素解析などの単語分割処理が必要無い。
- しくみが単純なので実装が簡単。
- 検索時に必ず元テキストが必要。WWW サーチエンジンには不向き。サイト内検索ならば問題無い。
- Suffix Array のサイズはインデックスポイントの数、つまりファイルサイズに比例する。

4.1.1 Suffix Array の仕組み

Suffix Array のしくみについて説明する前に、まず suffix について説明する。suffix とは検索対象となるテキスト中のある位置から始まりテキスト末尾までの範囲の文字列である。どの suffix も開始位置が特定されれば一意に決まる。この開始位置をインデックスポイント (index point) と呼ぶ [46]。例えば、テキスト「さくさくさくら」に対し、suffix「さくら」 は元テキストの 5 文字目から始まるのでインデックスポイントは 5 となる (図 4.1)。

このインデックスポイントの配列を、それぞれに対応する suffix の辞書順に従ってソートしたものが Suffix Array である。例えば、図 4.1 のインデックスポイントの配列「1 2 3 4 5 6 7」を対応する suffix でソートすると、「2 4 6 1 3 5 7」となる (図 4.2)。この配列が Suffix Array である。文献 [47] に非常にすっきりした Suffix Array の作成プログラム (sufsort.c) が掲載されているので図 4.3 にあげておく。

[1] 二分探索を用いるので、計算量は $O(\log(n))$: n = テキストサイズ。grep のようなテキスト全体を走査する方法では $O(n)$ かかる。巨大なファイルの検索において、その検索速度の差が顕著になる。

Text	さ	く	さ	く	さ	く	ら
Index point	1	2	3	4	5	6	7

図 4.1 インデックスポイント

文字列の検索には二分探索 (binary search) を用いる。図 4.4 に検索キー「くさくさ」で検索する例をあげる。二分探索は中心の要素と検索キーを比較して検索範囲を狭めていく探索手法である。まず、Suffix Array の中心 (4 番目) であるインデックスポイント 1 に対応する suffix「さくさくさくら」と検索キー「くさくさ」を比較する (phase 1)。辞書順で考えると「くさくさ」が小さいので、Suffix Array の中心より前半分 (1 番目から 3 番目) に検索範囲が絞られる。次に、その前半分の中心であるインデックスポイント 4 の suffix「くさくら」と検索キーを比較し (phase 2)、また検索範囲を絞る。このようにして、インデックスポイント 2 が最終的な検索結果となる (phase 3)。

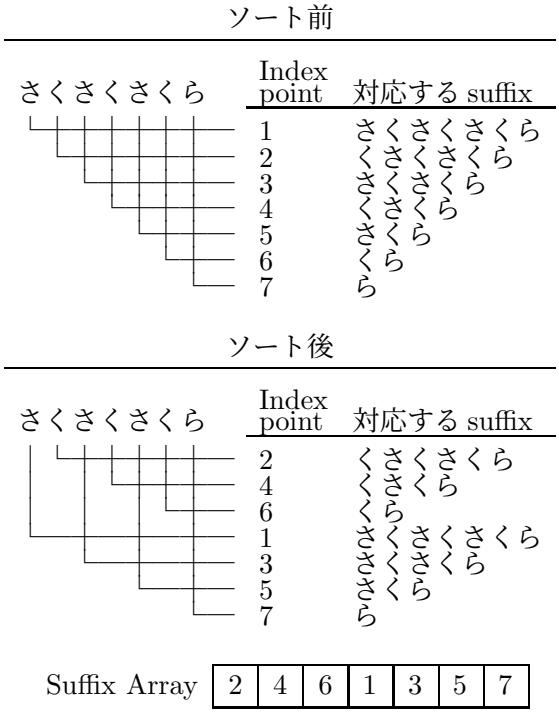


図 4.2 Suffix Array の作り方

```
#include <fcntl.h>
#include <malloc.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
/* usage: sufsort1 text > text.suf */

char *text;

suffix_compare(int *a, int *b)
{
  return strcmp(text + *a, text + *b);
}

main(int ac, char **av)
{
  struct stat stat_buf;
  int N, i, *suf;
  FILE *fd = fopen(av[1], "r");
  fstat(fileno(fd), &stat_buf);
  N = stat_buf.st_size;
  text = (char *)malloc(N+1);
  fread(text, sizeof(char), N, fd);
  text[N] = 0; /* pad with null */

  suf = (int *)malloc(N * sizeof(int));
  for(i=0;i<N;i++) suf[i] = i;

  qsort(suf, N, sizeof(int), suffix_compare);
  fwrite(suf, N, sizeof(int), stdout);
}
```

図 4.3 Suffix Array 作成プログラム

このように、Suffix Array は非常に単純な方法ということもあり、1970 年代にはすでに使われている [48]。1990 年になり Manber [18] により Suffix Array と命名された。Suffix Array についての教科書的な文献として [46] をあげておく。巨大な Suffix Array の作成方法、圧縮、高速化、正規表現、転置インデックスとの比較など、関連する話題が言及されている。また、日本語による簡単な解説として [49] がある。

Phase	Suffix array	対応する suffix
3 →	2	**くさくさ**くら
2 ⟶	4	くさくら
	6	くら
1 ⟶⟶	1	さくさくさくら
	3	さくさくら
	5	さくら
	7	ら

図 4.4 Suffix Array による文字列検索

4.1.2 Suffix Array による文字列検索ライブラリ SUFARY

Suffix Array を研究用に容易に利用できるように SUFARY を開発した。SUFARY は Suffix Array の作成・それを用いた検索を行なうライブラリで、フリーソフトである [20]。本論文では、2.3 節、3.3 節、4.2 節、4.3 節で使用している。

4.1.3 Suffix Array による疑似トライ

Suffix Array による疑似トライの実現方法について説明する。トライ構造の欠点であるデータ領域の非効率性を解消するための方法として有効である。従来のトライの実装に比べ必要なデータ領域は数分の 1 に圧縮できる。

まず、例として、テキスト BABAC に対する Suffix Array (図 4.5) を用いる。説明のために Suffix Array の各インデックスポイントの指す suffix を縦書きに表示した。ここで、ABAC, AC の先頭の A、BABAC, BAC の先頭の B と 2 文字目の A をそれぞれまとめて一つのノードとみなせば図 4.6 と同等なトライ (Suffix Tree) となることに気付くであろう。

つまり、先頭から X 文字目までが共通な suffix を指すインデックスポイントの Suffix Array での範囲をトライ上の深さ X のノードとして扱うことで疑似的なトライ構造が実現できる。この疑似トライにおいて、検索文字列に従ってノードを辿るという操作は、検索文字列の先頭文字から始めて一文字ずつ増やしながら (つまり、徐々に検索範囲を狭めながら) Suffix Array を検索するという操作になる。検索の度に徐々に検索範囲が狭められていくので検索効率は良い。

図 4.7 に疑似トライのアルゴリズムを示す。

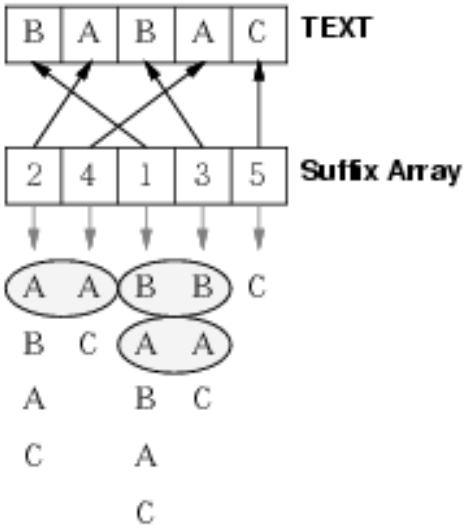
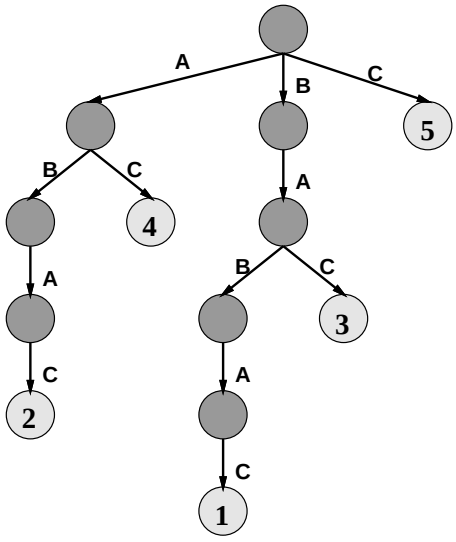


図 4.5 Suffix Array による疑似トライ



テキスト	B	A	B	A	C
インデックスポイント	1	2	3	4	5

図 4.6 Suffix Tree

- $t_1, ..., t_M$: 検索キーを構成する文字列
- M: 検索キーを構成する文字数
- $s_1, ..., s_N$: Suffix Array
- N: Suffix Array のサイズ

1. $l = 1, r = N$
2. for $(i = 1; i <= M; i++)$
   - Suffix Array の一部 (最初は Suffix Array 全体) $s_l, ..., s_r$ から、文字列 $t_1, ..., t_i$ を二分探索する。
   - 検索成功ならば、
     - 結果を l', r' に格納。($s_{l'}, ..., s_{r'}$ のそれぞれのインデックスポイントが指す文字列の先頭 i 文字は $t_1, ..., t_i$ 一致する。)
     - { トライの各ノードで行なう処理をここで行なう }
     - $l = l', r = r'$
   - 検索失敗なら i のループを抜ける

図 4.7 Suffix Array による疑似トライのアルゴリズム