Efficient Algorithms to Rank and Unrank Permutations in Lexicographic Order

Blai Bonet
Universidad Simón Bolívar
bonet@ldc.usb.ve

Introduction

- Ranking is the process of assigning a given permutation \( r \) over \( n \) numbers, a unique index (rank) \( r(\pi) \in \{0, \ldots, n!-1\} \)
- Unranking is the reverse process: given a rank \( r \in \{0, \ldots, n!-1\} \), find a permutation \( \pi \) such that \( r(\pi) = r \)
- Ranking is used to compute the index for PDB lookups, and is the most critical operation during node generation
- Ranking/unranking algorithms are said to be in lexicographic order if lexicographically consecutive permutations are ranked into consecutive integers. Thus, e.g., the identity permutation is ranked into 0 and the reverse permutation is ranked into n! - 1
- It has been argued that a lexicographic ranking can be more efficient for search as it tend to generate references (PDB indices) with some degree of locality

Related Work

- Naive algorithms run in \( O(n^2) \) time
- Knuth (1974) presents \( O(n \log n) \) lexicographic algorithms based on modular arithmetic
- Using a data-structure of Derta (1989), the number of operations can be reduced to \( O(n \log n) / \log \log n \) yet the implementation is complex and hidden constants are high
- Myrvold and Ruskey (2001) present non-lexicographic algorithms that run in linear time (these are commonly used in heuristic search)
- Korf and Schultze (2005) present linear time lexicographic algorithms that require a precomputed table of exponential size
- Korf and Schultze’s algorithms are non-uniform

Contribution

- Novel lexicographic algorithms that run in \( O(\log n) \) time
- The algorithms are simple and efficient and, in our experiments, ran faster than the linear time algorithms of Myrvold and Ruskey for permutations up to size 128, and fell a bit short for sizes up to 1,024
- Also, give novel non-uniform lexicographic algorithms that require much less space than Korf and Schultze’s

Non-Uniform Algorithms

Korf and Schultze’s Ranking Algorithm:

\[
\pi = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r)
\]

where \( \pi(r) \) is the number of elements to the left of position \( r \) that are less than \( r \)

\[
\pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r)
\]

In the example:

\[
\pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r)
\]

New Algorithm:

- Replace table \( T_r \) of size \( O(2^n \log n) \) bits by a single table \( T^*_r \) of size \( O(2^n \log n) \) bits, and perform \( \{n/m\} \) lookups instead of one.

For example, for \( m = 4 \), we have that

\[
T^*_r = T^*_r[a \rightarrow N \ll N] + T^*_r[a \rightarrow N \ll N] + T^*_r[a \rightarrow N \ll N] + T^*_r[a \rightarrow N \ll N]
\]

Therefore,

\[
\pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r)
\]

In the example:

\[
\pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r) = \pi(r) \cdot \pi(r)
\]

Interesting Tradeoffs:

Experiments

[Graphs showing ranking and unranking algorithms]