Proceedings of the Federated Conference on Computer Science and Information Systems pp. 429–432
DOI: 10.15439/2017F117
ISSN 2300-5963 ACSIS, Vol. 11

Using branching-property preserving Prüfer Code to encode solutions for Particle Swarm Optimisation

Hanno Hildmann
Universidad Carlos III de Madrid (**UC3M**)
Av. Universidad, 30 - 28911 Leganés - Spain
Email: hanno@cypherpunx.org / hanno.hildmann@uc3m.es

Dymitr Ruta
Emirates ICT Innovation Centre (**EBTIC**)
P.O. 127788 Abu Dhabi, UAE
Email:dymitr.ruta@kustar.ac.ae

Dina Y. Atia
Khalifa University of Science and Technology
P.O. 127788 Abu Dhabi - UAE
Email: dina.atia@kustar.ac.ae

A. F. Isakovic
Khalifa Semiconductor Research Center (**KSRC**),
Khalifa University of Science and Technology
P.O. 127788 Abu Dhabi - UAE
Email: iregx137@gmail.com / abdel.isakovic@kustar.ac.ae

***Abstract*—In the area of applied optimisation, heuristics are a popular means to address computational problems of high complexity. Modelling the problem and mapping all variations of its solution into a so-called *solution space* are integral parts of this process. Representing solutions as graphs is common and, for a special type of graph, *Prüfer Code* (PC) offers a computationally efficient mapping (algorithms of $\Theta(n)$-complexity are known) to $n-2$ dimensional Euclidean space. However, this encoding does not preserve properties such as e.g. locality and therefore PC has been shown to be a bad choice for entire classes of problems. We argue that PC does allow the preservation of some properties (e.g. degree of branching and branching vertices) and that these are sufficiently relevant for certain types of problems to motivate encoding them in PC. We present our investigations and provide an example where PC has been shown to be a useful encoding.**

I. Introduction & Outline

HEURISTICS (from the Greek *ευσίρκω*: *"to find"*, *"to discover"*) are approaches that *find* or *estimate* good solutions to problems, as opposed to reliably determining the best one. For the more complex problems it is often impossible to exhaustively check all possible solutions, motivating the use of a heuristic. Furthermore, many problems require only a certain quality of the solution, and investing resources in improving a solution past this point does not add any benefit.

In one way or another, heuristics use some underlying properties of the solution space to navigate it. This process is *iterative*: heuristics identify acceptable solutions and then continuously try to improve on them in some informed manner.

In order to be able to *move* from one solution to a better one, there has to be some relation between them. Using this relation enables the heuristic to estimate which alternatives to consider (so as to avoid having to consider them all).

Modelling a problem and *encoding* its solutions (i.e. the mapping into a domain) are important decisions in the process. There are many ways to represent solutions and we will only focus on one: graphs, and in our case, simple, undirected, connected and acyclic graphs, commonly called *trees* [5]. In §**II** we provide some background on trees and discuss known complexity results as well as a specific encoding that allows us to represent trees as unique sequences of numbers: *Prüfer Code* [15].

There is evidence from the literature that mapping a tree to Prüfer Code fails to preserve certain properties, which have been shown to be important for a number of meta-heuristics [10]. We take a closer look at which properties are indeed preserved and then argue in §**III** that for a certain class of problems the preserved properties are actually sufficient to motivate the use of Prüfer Code. We support this in §**IV** by referencing to our work, which successfully used Prüfer Code.

II. Graphs

A. Graphs and trees

A *graph* G is a pair $G = (V, E)$ of two sets: the set $V = \{v_1, \ldots, v_n\}$ of n *vertices* (which are also often referred to as *nodes* or *worlds*) and the set $E = \{e_1, \ldots, e_m\}$ of m *edges* (often called *lines* or *connections*). Each edge e_i is a tuple of two vertices, representing the two vertices that this edge connects (cf. [8], [3]). One sub-category of graphs are *connected graph without cycles* (i.e. the number of edges is $n-1$ for n vertices), commonly called *trees* [14]. Trees are graphs in which any two vertices are *connected* to each other by a finite path which can not contain cycles. Phrasing it like this makes it intuitively clear why this type of graph can represent a solution to e.g. decision trees or routing problems.

We distinguish vertices that are single end nodes (i.e. *leafs* in the tree) and those that are not (i.e. *branching points*).

B. Complexities of graphs

Given a set of n vertices, [4] showed that the family of different trees that can be constructed over this set has n^{n-2} members. This result is commonly known as *Cayley's Theorem* due to [5] (cf. [6]). The first combinatorial proof provided for this theorem was provided by Prüfer [15] in 1918 [14] using a mapping that represented trees with n vertices as strings of length $n-2$ (cf. §II-C). By showing that this set of strings therefore had n^{n-2} members, Prüfer proved *Cayley's Theorem*.

Algorithm 1 Encoding a tree-graph to Prüfer Code (cf. [13])

1: $L \leftarrow$ leaves of T
2: **for** $i \leftarrow 1$ to $(n-2)$ do **do**:
3: $\quad v \leftarrow$ node removed from the head of L
4: $\quad PC\,[i] \leftarrow$ neighbour of v
5: $\quad$ delete v from T
6: $\quad$ **if** $deg(PC\,[i]) = 1$ **then**
7: $\quad\quad$ add $PC\,[i]$ to L

Algorithm 2 Decoding Prüfer Code to a tree-graph (cf. [13])

1: $L \leftarrow$ nodes that do not appear in the Prüfer Code PC
2: **for** $i \leftarrow 1$ to $(n-2)$ do **do**:
3: $\quad v \leftarrow$ node removed from the head of L
4: $\quad$ add edge $\{v, PC\,[i]\}$ to T
5: $\quad$ **if** i is the rightmost position of v in PC **then**
6: $\quad\quad$ add v to L
7: $\quad v \leftarrow$ node removed from the head of L
8: $\quad$ add edge $\{v, PC\,[n-2]\}$ to T

If we restrict the branching factor for any vertex in the tree to a constant k, we get k-ary trees, which have been studied in the literature extensively [16], [9], [7]. The relation between *leafs* (n_l) and *branching* vertices n_b in a k-ary tree is $n_l = n_b(k-1)+1$ [16].

C. Encoding graphs as Prüfer Code (PC)

In addition to providing a proof to [5], Prüfer also provided us with an efficient mechanism to encode trees into sequences of $n-2$ integers (and back). Such $n-2$ dimensional Euclidean spaces are known to work well with swarm and evolutionary search algorithms and are therefore of potential interest to us.

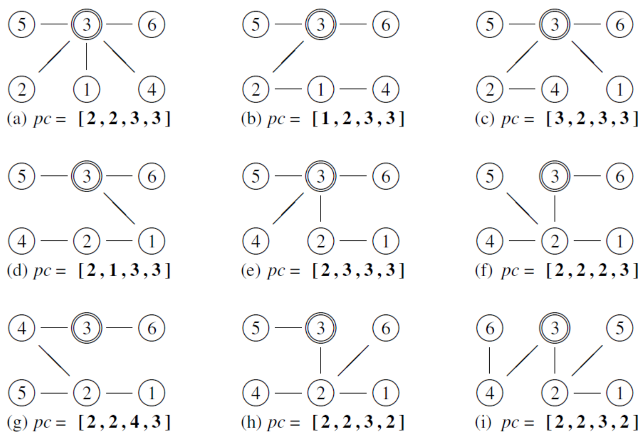

Fig. 1. The Prüfer codes similar to [2,2,3,3]. All variations (b) to (i) differ from the original string (a) in only one digit and the difference between that digit and the original is $|1|$; the root vertex v_3 is denoted by a double circle.

The specific way in which Prüfer Code (PC) is generated can result in fundamentally different trees being represented as very similar PCs [10] (see Figure 1, above). This is one of the likely sources of problems in the context of using PCs for heuristics, and we address this issue in in §III.

1) Algorithms: PC encoding and decoding follows a simple linear algorithm (cf. Alg. 1, Alg. 2, respectively), details of which can be found in [12]. From e.g. [13] we know that there are $\Theta(n)$-complexity algorithms (i.e. algorithms that can perform the translation either way in linear time) to do this.

Note that Alg. 1, above, assumes that the leaves are stored in a list (initially sorted in ascending order).

2) Solution space: Let's consider trees with n nodes (labelled 1 to n), resulting in PCs with $n-2$ positions. We use $\mathbb{PC} = \{pc_1, \ldots, pc_{n^{(n-2)}}\}$ to denote the set of all possible PC that meet this description. Clearly, any $\mathbb{PC}$ can be mapped into a subset of $\mathbb{N}^+$ by reading individual pc_i as a number (e.g. for $n=7$: this is $\{11111, \ldots, 26416, 26417, 26421, \ldots, 77777\}$). We use a PC's position in this set as the its ID (see example).

When exploring the solution space with heuristics we want there to be some correlation between a solution's location that space and its performance value. If we require that *similar* PCs represent trees encoding families of solutions (with regard to certain properties), we have to consider how we define *similar*.

Example: Let's consider encoding cooking recipes as trees (representing the order and inter-dependency of individual steps, started with step v_1). For a recipe with 7 steps, this can be represented as a tree with 7 nodes (of which there are exactly 16807 unique variations), each corresponding to exactly one PC with 5 positions. If the interpretation of *similar* is numerical distance between two codes (e.g. 24617 is followed immediately by 26421, cf. Figure 2 bottom row) then very similar PCs encode substantially different trees (see Figure 2). As pointed out in [10] this will make PC a sub-optimal choice for interpretations of similarity.

While the variations shown in Fig. 2 differ, they do not differ dramatically. This loose similarity was already enough to produce results of sufficient quality when we used PC to encode solutions representing cable diagrams [1], [2], [11].

III. Navigating Prüfer Code

A. A property-preserving mapping of PC to a solution space

The way trees are constructed from PC (cf. Alg. 2) implies that the connectivity of a vertex (the number of vertices it is connected to) is equal to the number of its occurrences in the PC + 1. This also means that not occurring vertices are leafs.

However, the positions of the integers matter, and exchanging two integers can result in more than the exchange of the corresponding vertices in the tree (see the example in Fig. 2).

1) Filtering PC: Using the above insight we look at certain filters for PCs that characterise properties of interest to us.

These filters, are defined with respect to a specific $pc_i \in \mathbb{PC}$:

- $\mathbb{I}_{pc_i}$, the set of all different integers that occur in pc_i
- $\mathbb{I}^+_{pc_i}$, the ordered list of all the occurring integers

Example: for trees with $n=5$, $\mathbb{PC} = \{[1,1,1], \ldots, [5,5,5]\}$; for e.g. $pc_i = [1,2,1]$: $\mathbb{I}_{[1,2,1]} = \{1,2\}$ and $\mathbb{I}^+_{[1,2,1]} = \{1,1,2\}$.

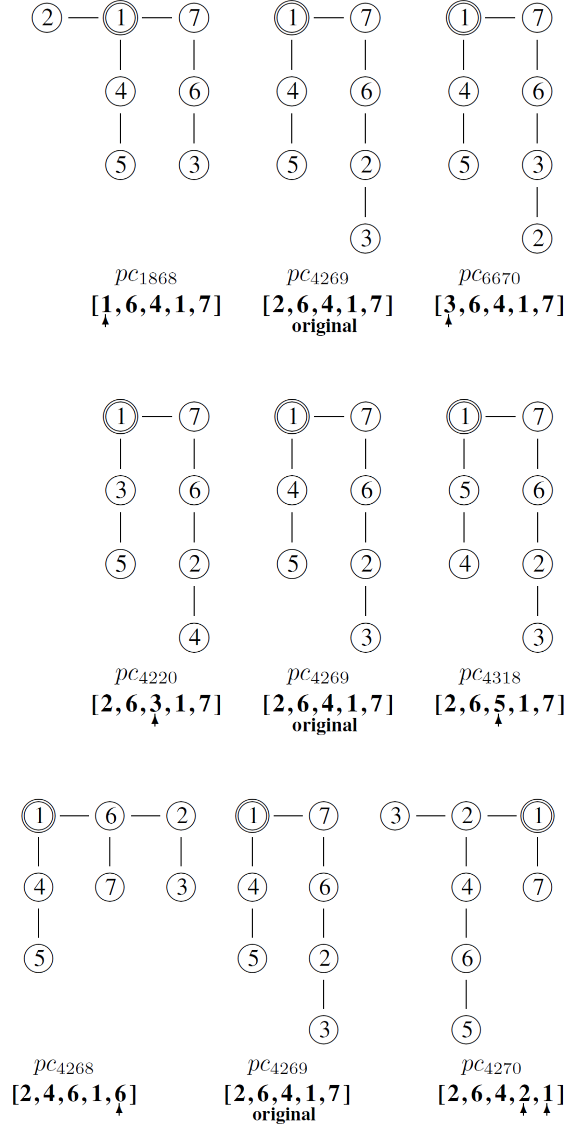

Fig. 2. Variations on pc_{4269}. An original graph (middle) is flanked by variations: (**first row**) differing in one position and just by 1 from the original integer (solution space $(n-2)$-dimensional) or (**second row**) the previous and the next ID (solution space: $\mathbb{N}^+$); the root vertex is assumed to be v_1.

2) Similarity classes: We use these filters to define the similarity classes $\mathbb{PC}_{\mathbb{I}}$ and $\mathbb{PC}_{\mathbb{I}^+}$, i.e. the subsets of $\mathbb{PC}$ where all members pc_j have the same $\mathbb{I}_{pc_j}$ or $\mathbb{I}^+_{pc_j}$, respectively:

- $\mathbb{PC}_{\mathbb{I}_{pc_i}}$, the subset of $\mathbb{PC}$ in which members are constructed using only the integers found in pc_i, and
- $\mathbb{PC}_{\mathbb{I}^+_{pc_i}}$, where all members have exactly the same integers as pc_i, but not necessarily in the same order.

$\forall pc_j \in \mathbb{PC}_{\mathbb{I}_{pc_i}} : \mathbb{I}_{pc_i} = \mathbb{I}_{pc_j}$ and $\forall pc_k \in \mathbb{PC}_{\mathbb{I}^+_{pc_i}} : \mathbb{I}^+_{pc_i} = \mathbb{I}^+_{pc_k}$ with $pc_i \in \mathbb{PC}_{\mathbb{I}_{pc_i}}$, $pc_i \in \mathbb{PC}_{\mathbb{I}^+_{pc_i}}$ and $\mathbb{PC}_{\mathbb{I}^+_{pc_i}} \subset \mathbb{PC}_{\mathbb{I}_{pc_i}}$.

Example: for $\mathbb{I}_{[1,2,1]} = \{1, 2\}$ and $\mathbb{I}^+_{[1,2,1]} = \{1, 1, 2\}$ we get: $\mathbb{PC}_{\mathbb{I}_{[1,2,1]}} = \{[1, 1, 2], [1, 2, 1], [1, 2, 2], [2, 1, 1], [2, 1, 2], [2, 2, 1], [2, 2, 2]\}$ and $\mathbb{PC}_{\mathbb{I}^+_{[1,2,1]}} = \{[1, 1, 2], [1, 2, 1], [2, 1, 1]\}$.

3) Distance: To create - individually for each pc_i - relative pc_i-solution spaces based on $\mathbb{I}_{pc_i}$ or $\mathbb{I}^+_{pc_i}$ we need to define a distance between pc_i and any pc_j in $\mathbb{PC}_{\mathbb{I}_{pc_i}}$ and $\mathbb{PC}_{\mathbb{I}^+_{pc_i}}$. Clearly the distance to itself ($pc_j = pc_i$) is zero.

We may either want a to define a single neighbour, a certain number of neighbours or sets of neighbours (potentially of varying sizes). This will directly impact the dimensionality of our solutions space: with a single neighbour we can use $\mathbb{N}^+$ as solution space, otherwise our solution space is n-dimensional or, in case of sets, of varying dimensionality. After defining a function to determine either a fixed number or a set of immediate neighbours of pc_i we can calculate the distance $\delta(i, j)$ between any two pc_i and pc_j as the shortest path connecting these two through their neighbours.

Example: for both $\mathbb{PC}_{\mathbb{I}_{pc_i}}$ and $\mathbb{PC}_{\mathbb{I}^+_{pc_i}}$ neighbourhood could (the choice is problem specific) be defined as, e.g.:

- the element in the respective set that is numerically the closest to pc_i (reading e.g. $[1, 3, 2, 4]$ as 1324), or
- all those elements that are created by exchanging two neighbouring digits of the pc, e.g. for $pc_i = [1, 2, 3, 4]$ this would be $[2, 1, 3, 4]$, $[1, 3, 2, 4]$ and $[1, 2, 4, 3]$.

B. Motivation

When optimising cabling structures for e.g. distributed antenna systems or routing network trees, the number of used *splitters* or *routers* (corresponding to branches in the tree) is an important factor as hardware plays a major role in the overall cost. In problems of this type constraints are commonly imposed on all paths from the root to the leaf nodes of the trees (e.g. power attenuation due to cable length which must not exceed a certain value); due to this variations over a fixed set of routers or splitters need to be explored.

On the other hand, having identified nodes in the network that exhibit high potential to become branches we want to consider changing their branching factors (i.e. the equivalent of replacing a splitter with a larger or a smaller one).

Specifically, our subsets of $\mathbb{PC}$ allow us the following:

1) $\forall pc_i \in \mathbb{PC}_{\mathbb{I}^+_{pc_{original}}}$: pc_i preservers the number of branching nodes, their branching degree as well as which node has how many branches. Only the specific allocation of leafs to these branches changes, as well as how these branching nodes are connected to each other.
2) $\forall pc_j \in \mathbb{PC}_{\mathbb{I}_{pc_{original}}}$: contrary to the above, pc_j does not ensure that the number of nodes with a certain branching degree stays the same, i.e. while the branching nodes do not change, their degree might, as does (as above) which leafs / other branching nodes they connect to.
3) In addition to the two above, we can explore variations on $\mathbb{PC}_{\mathbb{I}_{pc_{original}}}$ and $\mathbb{PC}_{\mathbb{I}^+_{pc_{original}}}$ by replacing all occurrences of an integer with one that does not occur in the original, or by simply adding or removing integers. As shown in Figure 1, these are more dramatic changes.

IV. Proof of concept application

Despite the claims made in [10] we successfully used Prüfer Code encoding to optimise cabling to power indoor antenna systems for large buildings [1], where small instances of $n = 20$ already have $20^{20-2} = 2.62\times10^{22}$ possible connection trees, (cf. Figure 3). Our work, tested for problems of up to 100 floors, showed that using Particle Swarm Optimisation obtained good solutions in short time (minutes)[1]. We also used Genetic Algorithms (GA) which, although inferior to PSO, performed well, indicating that using PC was a feasible approach. Cf. [2] for an overview over the results.

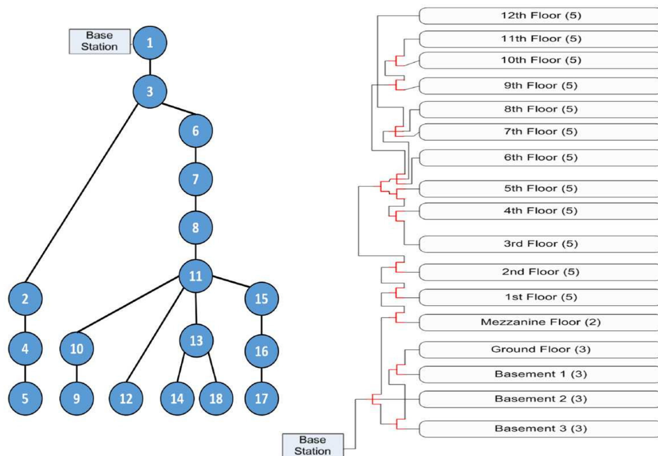

Fig. 3. An example solution for the Distributed Antenna Cabling Problem [1], [2], [11]. The objective is to connect all floors (and antennas on each floor) using splitters and cables, subject to power constraints imposed in the splitters and the antennas. The choice of branching nodes (and their degree) is a primary factor in this problem, making Prüfer Code a useful encoding.

A performance analysis of the algorithm showed that PSO converges towards good solutions. This is suggested by the fact that stagnating improvement over previous generations indicates approaching the best expectable solution (cf. Fig. 4). The argument is straight forward: if our exploration through PC-space were entirely random (and thus void of beneficial *similarities*) we would expect that the potential for finding improved solutions increased with additional searches, while the graph plotted in Figure 4 indicates the opposite.

V. Conclusion

Our investigations and the suggestions put forward in this paper do not refute the claims made in [10]. Instead, they are to be understood as an addition, in the sense that the we have identified a class of problems for which the encoding of trees in PC is beneficial. Specifically, when using trees to represent (a) variations on the branching of a tree (both in identifying the branching nodes we well as their degree of branching) and (b) the allocation of leaf nodes to branching nodes, Prüfer Code has proven to be a useful encoding. We intend to investigate this further by applying PC to other problems in the future.

[1]For comparison, a brute force search for $n = 8$ required 15 minutes of CPU time; our approach returned the same optimal result after 15 seconds.

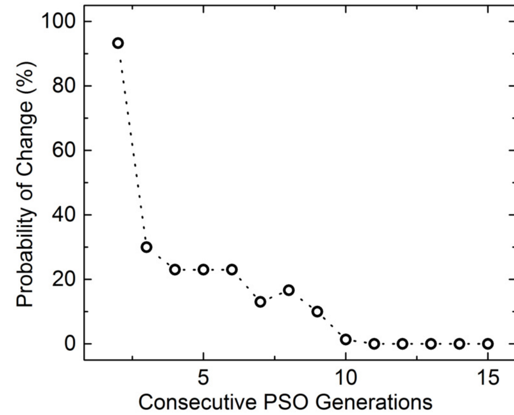

Fig. 4. The probability of finding a better solution plotted against the number of PSO generations resulting in unchanged best solution quality.

There have been other investigations into locality properties of PC (e.g. [14]) suggesting that the general results of [10] may not be all there is to PC. We have additional conjectures about this, which would require more space here and further investigations, and are outside the scope of this short paper.

References

[1] D. Y. Atia. Indoor distributed antenna systems deployment optimization with particle swarm optimization. M.Sc. thesis, Khalifa University of Science, Technology and Research, 2015.

[2] D. Y. Atia, D. Ruta, K. Poon, A. Ouali, and A. F. Isakovic. Cost effective, scalable design of indoor distributed antenna systems based on particle swarm optimization and prufer strings. In *IEEE 2016 IEEE Congress on Evolutionary Computation*, Vancouver, Canada, July 2016.

[3] P. Blackburn, M. deRijke, and Y. Venema. *Modal Logic*. Cambridge University Press, 2001.

[4] C. W. Borchardt. Über eine Interpolationsformel für eine Art symmetrischer Funktionen und über deren Anwendung. In *Math. Abh. Akad. Wiss. zu Berlin*, pages 1–20. Berlin, 1860.

[5] A. Cayley. On the theory of the analytical forms called trees. *Philosophical Magazine*, 13:172–6, 1857.

[6] A. Cayley. *A theorem on trees*, volume 13 of *Cambridge Library Collection - Mathematics*, pages 26–28. Camb. Univ. Press, July 2009.

[7] S.-H. Cha. On complete and size balanced k-ary tree integer sequences. *Int. J. of Applied Mathematics and Informatics*, 6(2):67–75, 2012.

[8] R. Diestel. *Graph Theory*. Elect. library of mathematics. Springer, 2006.

[9] S. K. Ghosh, J. Ghosh, and R. K. Pal. A new algorithm to represent a given k-ary tree into its equivalent binary tree structure. *Journal of Physical Sciences*, 12:253–264, 2008.

[10] J. Gottlieb, B. A. Julstrom, G. R. Raidl, and F. Rothlauf. Prüfer numbers: A poor representation of spanning trees for evolutionary search. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 343–350, San Francisco, California, 2001. Morgan Kaufmann Publishers.

[11] H. Hildmann, D. Y. Atia, D. Ruta, K. Poon, and A. F. Isakovic. *Nature-Inspired Optimization in the Era of IoT: Particle Swarm Optimization (PSO) applied to Indoor Distributed Antenna Systems (I-DAS)*, chapter tbd, page tbd. Springer, 2018 (forthcoming).

[12] B. A. Julstrom. Quick decoding and encoding of Prüfer strings: Exercises in data structures, 2005.

[13] P. Micikevičius, S. Caminiti, and N. Deo. Linear-time algorithms for encoding trees as sequences of node labels, 2007.

[14] T. Paulden and D. K. Smith. Developing new locality results for the Prüfer Code using a remarkable linear-time decoding algorithm. *The Electronic Journal of Combinatorics*, 14(1), August 2007.

[15] H. Prüfer. Neuer Beweis eines Satzes über Permutationen. *Archiv der Mathematik und Physik*, 27:742–744, 1918.

[16] P. V. Ramanan and C.L. Liu. Permutation representation of k-ary trees. *Theoretical Computer Science*, 38:83 – 98, 1985.