Algorithm Analysis with combstruct

The combstruct package is used to define and manipulate combinatorial structures. Once defined by a combstruct grammar, a structure can be counted, randomly generated, exhaustively generated, and its counting generating function can be found. In addition, a combstruct procedure can be created to describe an algorithm which acts on objects of that structure. There are tools available in the combstruct package to find the complexity descriptor generating function of the algorithm.

If f is a description of a procedure which acts on a combinatorial structure type A, then the complexity descriptor generating function is

$$\tau_f(z) = \sum_{n=0}^{\infty} c_n z^n$$

where c_n counts the total cost of operations required when f is applied to every object of size n from A.

These methods can be applied to a wide range of problems, including regular languages, context-free languages and combinatorial problems. Examples include analyzing a function over a regular expression, Pollard's rho-method for integer factorization, pathlength of various kinds of trees, strategies for binary powering, and mutually recursive functions.

For instructions regarding how to define a combinatorial structure see combstruct[specification]. For writing procedure descriptions, see combstruct[prog_specification]. An older introduction to the combstruct package, a guide to writing grammar specifications, and examples of finding the generating functions associated with the grammar specifications are all available in example worksheets.

```
> with(combstruct);
```

[*allstructs*, *count*, *draw*, *finished*, *gfeqns*, *gfseries*, *gfsolve*, *iterstructs*, *nextstruct*, *prog_gfeqns*, *prog_gfseries*, *prog_gfsolve*]

Examples

Binary Powering

As a simple example, consider the problem of binary powering. The goal is to compute x^k using few operations. The key is to write k in binary notation. For example x^{29} can be written as $x^{16} x^8 x^4 x$, so its calculation involves only a few multiplication and squaring operations. This method is particularly useful in cases where the multiplication operation is expensive, such as when x is a matrix. We find the total number of multiplication and squaring operations needed to compute x^k when k is a random integer whose binary representation is of length n.

To define this problem, we start with a grammar to describe k. The integer k is represented in its binary represenation as a sequence of ones and zeros.

```
> sys :=
  {chain=Sequence(bit),bit=Union(zero,one),zero=Atom,one=Ato
  m}:
```

We will work with unlabelled atoms.

```
> typ := unlabelled:
```

Then we describe the algorithm to compute x^k:

```
> binpower := proc(k::chain)
  local c1,b;
      if Type(k, Epsilon) then  # k=0
           nil
      elif Typematch(k, specfunc(b::bit,
  c1::anything,Sequence))
        then
          if Type(b,zero) then  # k is even
              # binpower(x,iquo(k,2))^2
              binpower(c1);
              squaring;
          elif Type(b,one) then  # k is odd
              # binpower(x,iquo(k,2))^2 * x
              binpower(c1);
              squaring;
              multiply;
          fi;
      fi;
  end:
```

We define the cost of the basic statements used in the algorithm by giving them complexity measures.

```
> measures := {squaring = 1, multiply = 1}:
```

Now we look for the generating functions. The function combstruct[prog_gfsolve] finds exact expressions for the counting generating functions of the non-terminals of the grammar and the complexity descriptors of the program descriptions.

```
> prog_gfsolve([sys,typ,{'binpower'=binpower}, measures],
  z);
```

$$\{\operatorname{one}(z) = z,\ \operatorname{zero}(z) = z,\ \tau_{binpower}(z) = 3\,\frac{z}{1-4\,z+4\,z^2},\ \operatorname{chain}(z) = -\frac{1}{-1+2\,z},\ \operatorname{bit}(z) = 2\,z\}$$

The truncated series of these generating functions can also be found using the function combstruct[prog_gfseries]. The order of the series is based on the Order environment variable.

```
> Order := 15:
> prog_gfseries([sys,typ,{'binpower'=binpower}, measures],
  z);
```

table([

$\operatorname{zero}(z) = z$

$\operatorname{bit}(z) = 2\,z$

$\operatorname{one}(z) = z$

$$\operatorname{chain}(z) = 1+2\,z+4\,z^2+8\,z^3+16\,z^4+32\,z^5+64\,z^6+128\,z^7+256\,z^8+512\,z^9+1024\,z^{10}+2048\,z^{11}+4096\,z^{12}+8192\,z^{13}+16384\,z^{14}+\mathrm{O}(z^{15})$$

$$\tau_{binpower}(z) = 3\,z+12\,z^2+36\,z^3+96\,z^4+240\,z^5+576\,z^6+1344\,z^7+3072\,z^8+6912\,z^9+15360\,z^{10}+33792\,z^{11}+73728\,z^{12}+159744\,z^{13}+344064\,z^{14}+\mathrm{O}(z^{15})$$

])

This result means that there are 8 chains of length 3, and a total of 36 multiplications and squarings are needed to apply the function binpower to all 8 chains. Thus the average number of operations on 3-bit long exponents is 36/8 = 4.5.

The function combstruct[prog_gfeqns] returns the system of generating function equations for this problem. It can be particularly useful on those occasions where the solver is unable to find an exact expression for the generating functions.

```
> prog_gfeqns([sys,typ,{'binpower'=binpower}, measures], z);
```

$$\left[\tau_{binpower}(z) = \tau_{binpower}(z)\,\mathrm{zero}(z) + \mathrm{zero}(z)\,\mathrm{chain}(z) + \tau_{binpower}(z)\,\mathrm{one}(z) + 2\,\mathrm{one}(z)\,\mathrm{chain}(z),\right.$$
$$\left.\mathrm{one}(z) = z,\ \mathrm{zero}(z) = z,\ \mathrm{bit}(z) = \mathrm{zero}(z) + \mathrm{one}(z),\ \mathrm{chain}(z) = \frac{1}{1 - \mathrm{bit}(z)}\right]$$

Obviously this algorithm can be improved, for instance by treating the case $k = 1$ specially. The new generating functions will reflect these improvements and in this way the performance of different algorithms can be compared.

Tree size and pathlength

The average path length of different kinds of trees can be explored with combstruct. The path length of a tree will affect the costs of search and insertion algorithms. We will examine the average internal path length of non-planar labelled trees where each node has either zero or two children.

```
> sys := {Tree=Union(node, tree), tree=Prod(node,tree2),
  tree2=Set(Tree,card=2), node=Atom}:
  typ := labelled:
```

The path length algorithm will make use of this procedure to find the tree size (number of nodes). It counts one for a node, and then recursively traverses the subtrees.

```
> treesize := proc(t::Tree)
  local s, x;
      if Type(t,node) then
          count
      elif Type(t, tree) then
          if Typematch(t, specfunc(node, s::anything,Prod))
  then
                  count;
              for x in s do
                  treesize(x)
              od
          fi;
      fi;
  end:
```

The internal pathlength is the sum of the distances of all nodes to the root, computed recursively as the size plus the pathlength of the subtrees.

```
> pathlength := proc(t::Tree)
  local s, x;
      if Type(t,node) then
          nil
      elif Type(t, tree) then
          if Typematch(t, specfunc(node, s::anything,Prod))
  then
              for x in s do
                  treesize(x);
                  pathlength(x)
              od
          fi;
```

```
    fi;
end:
```

A cost of 1 is assigned to each count operation.

```
> measures := {count=1}:
```

Then the same functions as before can be used to find the generating functions.

```
> prog_gfsolve([sys,typ,{'treesize'=treesize,'pathlength'=pathlength}, measures], x);
```

$$\left\{\mathrm{node}(x)=x,\ \mathrm{Tree}(x)=\frac{1}{2}\frac{\%1}{x},\ \mathrm{tree2}(x)=-\frac{-\frac{1}{2}\frac{\%1}{x}+x}{x},\ \mathrm{tree}(x)=\frac{1}{2}\frac{\%1}{x}-x,\right.$$

$$\left.\tau_{treesize}(x)=\frac{1}{2}\frac{\%1}{x\sqrt{1-2x^2}},\ \tau_{pathlength}(x)=2\,\frac{-\frac{1}{2}\frac{\%1}{x}+x}{-1+2x^2}\right\}$$

$$\%1 := 2-2\sqrt{1-2x^2}$$

```
> prog_gfseries([sys,typ,{'treesize'=treesize,'pathlength'=pathlength}, measures], x);
```

table([

$$\tau_{treesize}(x)=x+\frac{3}{2}x^3+\frac{5}{2}x^5+\frac{35}{8}x^7+\frac{63}{8}x^9+\frac{231}{16}x^{11}+\frac{429}{16}x^{13}+\mathrm{O}(x^{15})$$

$$\mathrm{node}(x)=x$$

$$\tau_{pathlength}(x)=x^3+3x^5+\frac{29}{4}x^7+\frac{65}{4}x^9+\frac{281}{8}x^{11}+\frac{595}{8}x^{13}+\mathrm{O}(x^{15})$$

$$\mathrm{tree2}(x)=\frac{1}{2}x^2+\frac{1}{2}x^4+\frac{5}{8}x^6+\frac{7}{8}x^8+\frac{21}{16}x^{10}+\frac{33}{16}x^{12}+\frac{429}{128}x^{14}+\mathrm{O}(x^{16})$$

$$\mathrm{Tree}(x)=x+\frac{1}{2}x^3+\frac{1}{2}x^5+\frac{5}{8}x^7+\frac{7}{8}x^9+\frac{21}{16}x^{11}+\frac{33}{16}x^{13}+\mathrm{O}(x^{15})$$

$$\mathrm{tree}(x)=\frac{1}{2}x^3+\frac{1}{2}x^5+\frac{5}{8}x^7+\frac{7}{8}x^9+\frac{21}{16}x^{11}+\frac{33}{16}x^{13}+\mathrm{O}(x^{15})$$

])

Because we are working in the labelled universe, the generating functions are exponential generating functions. Thus there are 5! * 1/2 = 60 trees with 5 nodes, and the total pathlength of those 60 trees is 5! * 3 = 360, so the average pathlength of a tree with 5 nodes is 360/60=6.

We can re-define our tree so that each node has exactly 0 or 3 children, and then reuse the same algorithms.

```
> sys := {Tree=Union(node, tree), tree=Prod(node,tree2),
  tree2=Set(Tree,card=3), node=Atom}:
```

```
> prog_gfseries([sys,typ,{'treesize'=treesize,'pathlength'=pathlength}, measures], x);
```

table([

$$\tau_{treesize}(x)=x+\frac{2}{3}x^4+\frac{7}{12}x^7+\frac{5}{9}x^{10}+\frac{715}{1296}x^{13}+\mathrm{O}(x^{16})$$

$$\mathrm{node}(x)=x$$

$$\tau_{pathlength}(x) = \frac{1}{2}x^4 + \frac{3}{4}x^7 + \frac{23}{24}x^{10} + \frac{251}{216}x^{13} + O(x^{16})$$

$$\mathrm{tree2}(x) = \frac{1}{6}x^3 + \frac{1}{12}x^6 + \frac{1}{18}x^9 + \frac{55}{1296}x^{12} + O(x^{15})$$

$$\mathrm{Tree}(x) = x + \frac{1}{6}x^4 + \frac{1}{12}x^7 + \frac{1}{18}x^{10} + \frac{55}{1296}x^{13} + O(x^{16})$$

$$\mathrm{tree}(x) = \frac{1}{6}x^4 + \frac{1}{12}x^7 + \frac{1}{18}x^{10} + \frac{55}{1296}x^{13} + O(x^{16})$$

])

References

The following are a list of references relevant to a previous incarnation of this program called Lamba-Upsilon-Omega, (Λ-Y- Ω) or LUO for short. They contain the theoretical foundations of this program and examples of other applications. The code for LUO, written in a combination of Maple and Caml, and electronic versions of these papers are available from http://www-rocq.inria.fr/algo/libraries/libraries.html.

- *Automatic average-case analysis of algorithms* P. Flajolet, B. Salvy and P. Zimmermann. Theoretical Computer Science, Series A, vol. 79, no. 1, 1991.
- *Lambda-Upsilon-Omega the 1989 cookbook* (INRIA Research Report 1073) P. Flajolet, B. Salvy and P. Zimmermann, 1989.
- *Lambda-Upsilon-Omega : an assistant algorithms analyzer* P. Flajolet, B. Salvy and P. Zimmermann. Applied Algebra, Algebraic Algorithms and Error-Correcting Codes. T. Mora (editor). Lecture Notes in Computer Science vol. 357, 1989.