6.871 Term Project Report

Velin K Tzanov

1. Motivation and Problem Statement

If we can help Math students learn how to integrate, by creating a knowledge-based system that does integration for them (and shows them how it did it), why can't we do the same with Computer Science students and dynamic programming (and possibly other algorithmic techniques)? Inspired by this question, I decided to design and implement a system that would take the description of an algorithmic problem and construct a solution (algorithm) for it. Here are three examples of such problems:

Example 1:
You are given a sequence of N numbers and an integer K. Find a split of this sequence into K sub-sequences of consecutive numbers, so that the product of the sums of the numbers in the groups is maximized.

Example 2:
You are given N cylinders and each one has specified height, weight and base radius. You can use those cylinders to build a tower, by placing them one over another. In doing that you have to keep in mind that a heavier cylinder cannot stand over a lighter one and that a larger (by base radius) cylinder cannot stand over a smaller one. What's the highest tower you can build, using the given set of N cylinders?

Example 3:
You are given the numbers from 1 to N and an integer P between 1 and N. The g-number of a sequence is defined as the number of monotonically decreasing sub-sequences in that sequence. For example in `5 2 4 9 7 3 1 10 8 6` the g-number is 4 (the monotonically decreasing sub-sequences are `(5 2) (4) (9 7 3 1) (10 8 6).` Find the number of permutations of the numbers from 1 to N that have a g-number equal to P.

2. Input Language

In order to describe the problem statements to my program, I had to create a simple, yet powerful language that could be used in my input files. Of course, ideally such a program would read the problem statements in natural language, but that is a really hard task, as we know. That is why I came up with a very simplified object-oriented language, inspired by Minsky's frames and the C/C++ `struct` keyword. In this language we have three types of entities: primitive variables types, collections and structures.

Primitive variable types are things like integers and floating point numbers. In the current version of the language there is only one primitive type: `INT`. Collections are simply groups of objects of the **same** type (either a primitive type, or a structure). At the moment I have two type types of collections: `SET` and `SEQUENCE`. The difference is that the latter is ordered, while the first one doesn't have any notion about order (which doesn't mean that you can't take its elements and arrange them in some order later).

Structures on the other hand are completely user-defined. They, similarly to the `struct`s in C/C++ are collections of objects, which can be of **different** types (primitive types or collections). The main difference with the collections described above (besides the fact that you can have different types of items) is that you have only a finite number of items, while in `SET`s and `SEQUENCE`s you have an unbounded number in items (i.e. you specify the type of the items and that's it, while in the structures you enumerate and name the items one by one). Here is an example of the definition of collections and structures for example problem 2 described above:

```
STRUCT CYL INT HEIGHT INT WEIGHT INT BRAD
STRUCT TOWER SET CYL POOL SEQUENCE CYL CHOSEN INT HEIGHT
```

The first line defines `CYL` as a structure that has an integer `HEIGHT`, an integer `WEIGHT` and an integer `BRAD`.
The second line defines `TOWER` as a structure that has a set of `CYL`s called `POOL`, a sequence of `CYL`s called `CHOSEN` and an integer `HEIGHT`.

In addition to that we can specify relations between the items in a structure using the `EQUATE` command. It takes the structure name as its first argument and then it takes the name of one of its items. The next argument is an operation that is performed on the following arguments, which are also items from the structure. For example if we want to make item `DEBT` of a structure `ACCOUNT` equal to the sum of all items in the `PAYMENTS` collection of `ACCOUNT`, we include the following line:

```
EQUATE ACCOUNT DEBT SUM PAYMENTS
```

In order to make the language more flexible we also have the `FIELD` keyword that can be used when describing an item. It means that we are not interested in the item whose name follows, but rather in an item of that item. Hence the parser looks for another name after the following name. Also, we have the `ELEMENT` keyword, which stands for an element of a collection (remember, items in collections are nameless because they are unbounded in number). This element is an abstract notion of an element in this group, not any particular element. Here is an illustration of these two keywords, which tells us that the height of a tower is equal to the sum of the heights of the cylinders in its chosen set:

```
EQUATE TOWER HEIGHT SUM FIELD CHOSEN FIELD ELEMENT HEIGHT
```

Finally, our language has commands that tell the program what objects are given to us and what objects are we trying to compute. There are the `GIVEN` command and the `FIND / HAVING` pair, which are illustrated below, again in the context of the tower of cylinders example:

```
GIVEN SET CYL INP
FIND TOWER POOL INP HAVING MAX HEIGHT
```

In addition to those, my initial version of the language had means to define functions, so that things like the g-number from example problem 3 above could be described. Reading and understanding those language constructs was never implemented however (for reasons explained in section 4 below), so I will just give a quick example of this feature, without much explanation. It simply says that we add 1 to `G` every time an item in the `BOARDS` sequence is higher than the previous item (something that people usually write as `G(X) = G(X-1) + (BOARDS(X) < BOARDS(X-1)) )`

```
EQUATE FENCE G RECURSIVE BOARDS ADD G -1 SMALLER ELEMENT -1 ELEMENT 0
```

This language seems to be powerful enough for the problems I tried my system on, though I have to admit that often it's not trivial to convert a problem statement into an input file. Nevertheless, it's almost always possible to do that and that was the goal of this project (ideally we would use natural language processing, but unfortunately we don't have this available yet).

3. System Architecture

Having this object-oriented input representation, the most natural way to organize the state of the program is to create instances of those objects, defined in the input file, in memory and assign various properties to those objects. Initially we have the given objects with the property that they're given, known, computed, whatever word you like. We also have the goal objects (the one defined with the `FIND` command) and they have the property that they are a goal. Then our system starts to reason in a rule-based manner. The knowledge base is a set of rules, which based on the given objects and their properties create other objects and/or assign new properties to objects. The execution of the program might terminate in two ways. If a goal object is computed it means we have a sequence of steps by which we can compute the goal from the initially given objects – these steps are the algorithm that our program will produce. The other condition for termination is the inability to trigger any of the rules, which would mean that our program cannot solve the given algorithmic problem.

Actually, this reasoning strategy is very appropriate for this problem not only because it's the most convenient for handling the input language. In fact, this is the type of reasoning that human experts do. Being such an expert myself, I can say that with certainty. Usually when given an algorithmic problem, one starts to think "Well, what can we infer from the input data?" Then at some point the thinking process switches backwards and one starts to think, "Well, what do I need in order to compute the desired output?" If the problem still cannot be solved, the mind goes back and forth between the two approaches, trying to intersect the two sets somehow. During the process, especially if it takes longer, a lot of new constructs (objects, in our language) are invented and a lot of hypotheses are formed, tested and proved or disproved. At the end, either an algorithm is found, or the problem is left open. Proofs of impossibility are extremely rare and they are very hard even for human experts.

Our system does a very good job at simulating the "two meeting sets" way of thinking and the format of the conclusion (i.e. it either says "Here's a solution" or "I don't know; there might be a solution, but I cannot find it"). However, it does a poor job in creating new constructs (objects) and forming/testing/proving hypotheses for reasons explained in the following section.

4. Hardness of the Problem

While I was building the knowledge base of the program I realized that there are actually two types of problems. Those of the first type only deal with combinatorial and logical terms like sets, subsets, sequences, orders, sorts, maximums, minimums, aggregates, etc. Those from the second type deal with more than that: they usually have meaningful numbers involved with them and they usually deal with functions over those numbers. In a sense you have to go out of the finite domain of sets, unions and aggregates and into the infinite domain of numerical functions.

Of course, the difference is not as big as the difference between finite and infinite and in fact you can use the same paradigm described in the previous section to solve both types of problems. The finite/infinite distinction I am writing about comes from the fact that in the first type of problems the things you can infer (or the rules that you have to write) are very limited in number. An example of such a rule is: if you have polynomial number of options for the elements of a set, then you have polynomial number of options for its aggregates, but exponential number of options for its subsets. The number of such rules is small simply because the combinatorial and logic terms are finite in number: set, sequence, union, intersection, product of the elements… you continue the list with some more terms and you're done. But when you enter the world of numerical functions you have a whole set of new dependencies, equations, deductions, proofs, etc. with every new input file that you get. And yes, you can write rules for each one of those proofs and deductions and have your program do them, but you will always be able to find new and new problems that your program cannot solve and that require new hypotheses and equations that your program has never seen.

This is best illustrated by an example. Let's consider our example problem 3 from Section 1 above. Here is the description of the problem in our input language. It has the simplest description among all dynamic programming problems I created an input file for:

```
STRUCT FENCE SEQUENCE INT BOARDS INT GRPS
EQUATE FENCE GRPS RECURSIVE BOARDS ADD GRPS -1 SMALLER
ELEMENT -1 ELEMENT 0
GIVEN SET INT INP
GIVEN INT P
FIND COUNT FENCE BOARDS SET2SEQ INP GRPS P
```

The description of the problem is short. The solution of the problem is also short. If we denote the answer to our problem by `A(N, P)`, we have the recurrent formula:

```
A(N, P) = P*A(N-1, P) + (N-P+1)*A(N-1, P-1)
```

From this recurrent formula on, the dynamic programming is trivial. So what we have to do is take this short and simple problem statement and come up with this short and simple recurrent formula. Doing that, however, is neither short, nor simple.

Here is how it works for humans. Let's consider all size-N permutations, which have a g-number equal to P. Consider the highest number in the permutation. It has N possible positions. Moreover, it is always the first number in its monotonically decreasing sub-sequence. What if we remove that number? We would end up with the same problem, but having size-N-1 permutation and g-number equal to P or P-1. We would get P if removing the number doesn't unite the two sub-sequences around it. There are P such positions in the permutation (those which are right before sub-sequences), thus if we multiply P by A(N-1, P) we would get the number of size-N permutations with g-number P, which match this case. In the other case, where removing the highest number does unite two sequences, we have N-P+1 possible positions for that number. Hence multiplying N-P+1 by A(N-1, P-1) will give us the number of sequences from this branch. Hence, the answer of A(N,P) is the sum of the two formulas above.

As you can see, the solution involves defining the recurrent function A(N, P), which would probably be easy for our program, since this is the function it has to compute. Then it has the idea of finding a recurrent formula, which could easily be hard-coded in our program, since this is the major paradigm of dynamic programming. Then it has to construct the recurrent formula in three steps. First, it has to consider the highest element and consider removing it (i.e. creating a new `FENCE` object, whose only difference with the given object is the lack of the highest element). These considerations could probably be rules in the knowledge base, so they're OK. But then the program has to compute the relation between the two values. It has to understand that there are two cases for how the removal of the highest element affects the result of the formula. It also has to consider each of the two options and come up with formulas that represent the result in each of those branches. It then has to weigh in the "probabilities" of entering those branches (i.e. the number of ways you can enter them) and multiply the recurrent formulas by these weights. This, I claim, is very hard for a rule-based system with a finite knowledge base. Writing rules for this kind of manipulations (taking a function that counts the number of times an element in a sequence is higher than its predecessor, subdividing that function into cases and computing and "proving" values for those cases) is possible, but it means writing very, very case-specific rules. And since the set of algebraic functions is infinite, the number of different manipulations and proofs one can come up with is also infinite and thus you will always have to come up with new and new rules in order to make your rule-based system work correctly… until you turn it into a lookup table.

The above is enough to discourage me from trying to attempt solving the second type of dynamic programming problems, but what actually made me give up was the following example. Let's consider this problem:

Example 4:
A bracket sequence (BS) is a sequence of bracket characters and it has two properties: length and depth. A valid BS is defined as follows:
- The empty sequence is a valid BS with length 0 and depth 0.
- If A is a valid BS, then (A) is a valid BS with length A.length+2 and depth A.depth+1
- If A and B are valid BS, then AB is a valid BS with length A.length + B.length and depth max(A.depth, B.depth)

Problem: find the number of different bracket sequences of given length and depth.

The solution to this problem is also a dynamic program, which employs a recurrent formula… but on another function. The trick to finding a good solution is to introduce a new function: incomplete BS (IBS). This is a BS, but it might have some open brackets, which aren't closed. These IBS have three properties instead of two: depth, length and number of open brackets outstanding. Defining a recurrent formula on the number of IBS with given values for these three parameters is much, much easier than doing it for the original function with two parameters.

But how do you get a computer program to invent a new, more complicated class, object or function and use it to solve a problem for the simpler version of that class, object or funtion?! I don't know. In fact even experts don't know how sometimes, because even they fail at solving these problems sometimes, especially this type, which requires more creativity.

Example problem 3 was given on a Bulgarian national competition in informatics for high school seniors (note: Bulgarians are good at this stuff) and only 18 out of the 54 contestants were able to solve it. Example problem 4 (the last one) was given to the top 10 Bulgarian high school juniors and only one of them was able to solve it correctly. This convinced me that solving this type of problems, especially those that require the creation of new classes of objects, is beyond the scope of knowledge-based, automatic dynamic programming solvers. In fact, solving these problems seems much closer to solving general math problems, then solving integration.

Given the above, I decided to limit my scope to the problems of the first type (those dealing only with set combinatorics and logic) and get rid of the recursive function definition in the `EQUATE` command that was used to describe the numerical functions of the problems of the second type.

5. Knowledge Base and Property Propagation

Having limited our scope to primitive variable types, collections and user-defined structures that encapsulate those primitive types and collections, we can go ahead and define the possible properties of the objects and the rules that propagate these properties. The main property of an object is whether it can be computed from the input data. In fact, it is not just whether it can be computed, but rather whether it can be computed in a reasonable amount of time. Since in theoretical computer science reasonable often means polynomial, I've decided to separate the degrees of computability of a given object into three classes: KNOWN, POLYSIZE and EXPONENTIAL. The first one is assigned to objects, which are either part of the input data, or are directly computed from it. POLYSIZE are objects for which we have polynomial number of options, based on the input data. For example if we are given a sequence and an object is a prefix of that sequence, we know that we have only a linear (i.e. polynomial) number of possibilities for what this prefix might be. The last degree of computability, EXPONENTIAL stands for objects for which we have exponential number of possibilities. For example a subset of a set (without any other constraints) is EXPONENTIAL even though the set might be KNOWN.

In addition to that property we have another property, which exists only for POLYSIZE objects. It represents something like an ID of the POLYSIZE set to which these objects belong. For example if you have sub-sequences of the same sequence they are all POLYSIZE objects. However, the union of POLYSIZE objects is EXPONENTIAL. Hence, you would generally not be able to compute any aggregate over those objects (like product of some of their properties). However, if these sub-sequences are consecutive sub-sequences of the same sequence (i.e. a split of that sequence), then they have the notion of order among one another. That is why, they are assigned a common group ID by the SPLIT operator. Then, when the PRODUCS operator (or some other aggregate with similar properties) takes these sequences, it will figure out that they have order among them and it will know that by dynamic programming and the divide & conquer technique in can compute the aggregate in polynomial time.

Now, having defined those, all we have to do is provide a knowledge base of operators, which can propagate those properties among objects, which are connected by collections and/or structures. Whenever an object with the GOAL property becomes POLYSIZE or KNOWN, we know that we can compute that object and hence we can compute the goal, which means success for our program.

A complete list of the knowledge base is in the appendix. The more simple rules are plain propagation like the PREFIX operator that takes a sequence and a number K and returns a size-K prefix of that sequence. This operator (rule) looks at the properties of the sequence and the integer K and if they both are POLYSIZE or better, it assigns the worst of the two values as the value for the prefix. Hence if the sequence is KNOWN, but K is POLYSIZE, the prefix is POLYSIZE. Also, if K is KNOWN, but the sequence is POLYSIZE, the prefix is also POLYSIZE. Of course, if both are KNOWN, the prefix is also KNOWN. The group ID of the prefix (if it's POLYSIZE) is also transferred from the sequence.

Examples of more complicated rules are the SPLIT and PRODUCT operators I described above. PRODUCT takes collections of elements, or multiple elements of collections (it works no matter whether you give it a collection or the elements of that collection) and marks the product as POLYSIZE if all of the elements are either KNOWN or POLYSIZE and if all of those that are POLYSIZE have the same group ID.

One last thing to be said about the rules and the propagation of properties is that the items of a collection or a structure automatically receive the properties of their collection/structure, if they are better than theirs. For example when a set becomes KNOWN, all of its elements become KNOWN as well, and so these elements (or some sorted subset of them) can be given to an aggregate like PRODUCT.

6. Reasoning Depth and Examples

While I've taken the easier half of dynamic programming problems that has only a handful of operators and rules, the depth of the reasoning is still infinite. For example if one takes two simple problems and makes the output of the first be the input to the second, this will make my program do twice as deep reasoning. If the user repeats this process again and again, we would end up with more and more reasoning, without any theoretical bound.

Note, however, that this doesn't mean our program will never stop. In the case where the program creates new objects this might be the case (because it might also create newer and newer objects out of the ones it has just created), but our program is limited only to the non-numerical, non-creative side of dynamic programming. Thus when it has a set of objects and it goes through all of them without propagating anything, it would just give up and tell the user that he/she has to solve the problem him/herself.

Here are the input files of the first two examples in Section 1, followed by the input file that combines them into one big problem (one that sets the number of sub-sequences in the first example to be equal to the height of the highest tower).

Example 1:
```
STRUCT GROUP SEQUENCE INT ELS INT SM
EQUATE GROUP SM SUM ELS
STRUCT DIV SEQUENCE INT BIG INT PARTS SET GROUP SMALL INT
BENEFIT
EQUATE DIV FIELD SMALL FIELD ELEMENT ELS SPLIT PARTS BIG
EQUATE DIV BENEFIT PRODUCT FIELD SMALL FIELD ELEMENT SM
GIVEN SEQUENCE INT INP
GIVEN INT K
FIND DIV BIG INP PARTS K HAVING MAX BENEFIT
```

Example 2:
```
STRUCT CYL INT HEIGHT INT WEIGHT INT BRAD
STRUCT TOWER SET CYL POOL SEQUENCE CYL CHOSEN INT HEIGHT
EQUATE TOWER CHOSEN SORTEDSUBSET POOL FIELD POOL FIELD
ELEMENT WEIGHT FIELD POOL FIELD ELEMENT BRAD
EQUATE TOWER HEIGHT SUM FIELD CHOSEN FIELD ELEMENT HEIGHT
GIVEN SET CYL INP
FIND TOWER POOL INP HAVING MAX HEIGHT
```

Composite Example:
```
STRUCT GROUP SEQUENCE INT ELS INT SM
EQUATE GROUP SM SUM ELS
STRUCT DIV SEQUENCE INT BIG INT PARTS SET GROUP SMALL INT
BENEFIT
EQUATE DIV FIELD SMALL FIELD ELEMENT ELS SPLIT PARTS BIG
EQUATE DIV BENEFIT PRODUCT FIELD SMALL FIELD ELEMENT SM

STRUCT CYL INT HEIGHT INT WEIGHT INT BRAD
STRUCT TOWER SET CYL POOL SEQUENCE CYL CHOSEN INT HEIGHT
EQUATE TOWER CHOSEN SORTEDSUBSET POOL FIELD POOL FIELD
ELEMENT WEIGHT FIELD POOL FIELD ELEMENT BRAD
EQUATE TOWER HEIGHT SUM FIELD CHOSEN FIELD ELEMENT HEIGHT

STRUCT COMPOSITE DIV A TOWER B SEQUENCE INT BIG INT BENEFIT
SET CYL POOL INT HEIGHT
EQUATE COMPOSITE FIELD A BIG EQUALS BIG
EQUATE COMPOSITE BENEFIT EQUALS FIELD A BENEFIT
EQUATE COMPOSITE FIELD B POOL EQUALS POOL
EQUATE COMPOSITE FIELD A PARTS MAX FIELD B HEIGHT
GIVEN SEQUENCE INT BIGINP
GIVEN SET CYL CYLINP
FIND COMPOSITE BIG BIGINP POOL CYLINP HAVING MAX BENEFIT
```

If you run the program on the above examples you will notice that it solves example 1 in 3 steps and example 2 in 2 steps. However it, takes 9 steps to solve the composite examples (it adds 4 steps to propagate the parameters and the results from one sub-problem to another). This matches our expectations of the potentially infinite reasoning depth.

7. Lessons Learned

Working on this project changed my mindset with respect to solving dynamic programming. I've always viewed it as applying a subset of a set of standard techniques, but now I realize that it cannot really be encapsulated in a set of techniques (rules). For example creating a new function (class of objects) that has more parameters than the original one is not something that can be encapsulated in any rule or piece of code of reasonable complexity (i.e. something of the order of the things we studied in this class). I can say that now I respect the field I'm good at much more than before.

Also, it wasn't before trying to create this automated dynamic programming solver that I realized there are actually three different types of dynamic programming: one which requires creativity; one that doesn't, but requires ability for algebraic manipulation and ability to prove hypotheses in this area; and a third type, which doesn't require any of these, but it can still capture many interesting problems that are challenging for some Computer Science students (as mentioned in the Motivation section on the top).

Finally, this project gave me valuable experience with creating rule-based systems, which taught me some things I couldn't get from the lectures and papers. For example, the fact that writing rules without having a nice, clean chart of what value of what property means what, messes things up completely and causes plenty of problems, I had to learn the hard way.

Lastly I'd like to thank Prof. Davis for the opportunity to proceed with such ambitious project. I really did learn a lot from it and I'm grateful that I was allowed to do so in this class.

Appendix A. Complete Knowledge Base

Here is a complete list of the knowledge base of my simplified dynamic programming solver. Please note that even though the paper mentions the possibility of having rules that transfer the GOAL property "backwards", I have not implemented such rule. The reason is that in our set of simplified dynamic programming problems the "forward" and "backward" rules are essentially the same – they only go in different directions.

Each of the rule descriptions below starts by a line having the name of the operator that the rule corresponds to, as well as the types of the domain and the range of the operator. Then follows a description of the logic of the rule (i.e. how the `know` and the `groupID` variables of the destination variable are computed from the `know` and the `groupID` variables of the source variables).

```
EQUALS (anything -> anything)
know(dest) = know(src)
groupID(dest) = groupID(src)
```

```
UNION (2 or more collections -> collection)
know(dest) = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then groupID(dest) = this value
else groupID(dest) = 0
```

```
INTERSECTION (2 or more collections -> collection)
know(dest) = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then groupID(dest) = this value
else groupID(dest) = 0
```

```
PREFIX (integer, sequence -> sequence)
know(dest) = smallest of know(src)
groupID(dest) = groupID(src_sequence)
```

```
SUFFIX (integer, sequence -> sequence)
know(dest) = smallest of know(src)
groupID(dest) = groupID(src_sequence)
```

```
SPLIT (integer, sequence -> set of sequences)
know(dest) = POLYSIZE
groupID(dest) = new UNIQUE_ID
```

```
SAMPLE (integer, set -> set)
know(dest.elements) = POLYSIZE
groupID(dest.elements) = new UNIQUE_ID


SORTEDSET (set, anything -> sequence)
know(dest) = know(src)
groupID(dest) = groupID(src)


SORTEDSUBSET (set, anything -> sequence)
know(dest.elements) = POLYSIZE
groupID(dest.elements) = new UNIQUE_ID


PRODUCT (1 or more collections -> int)
tmpKnow = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then tmpGroupID = this value
else tmpGroupID = 0
if (tmpKnow = KNOWN or (tmpKnown = POLYSIZE and tmpGroupID
> 0)) then {
    known[dest] = POLYSIZE
    groupID[dest] = tmpGroupID
}


SUM (1 or more collections -> int)
tmpKnow = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then tmpGroupID = this value
else tmpGroupID = 0
if (tmpKnow = KNOWN or (tmpKnown = POLYSIZE and tmpGroupID
> 0)) then {
    known[dest] = POLYSIZE
    groupID[dest] = tmpGroupID
}


COUNT (1 or more collections -> int)
tmpKnow = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then tmpGroupID = this value
else tmpGroupID = 0
```

```
if (tmpKnow = KNOWN or (tmpKnown = POLYSIZE and tmpGroupID
> 0)) then {
    known[dest] = POLYSIZE
    groupID[dest] = tmpGroupID
}


MIN (1 or more collections -> int)
tmpKnow = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then tmpGroupID = this value
else tmpGroupID = 0
if (tmpKnow = KNOWN or (tmpKnown = POLYSIZE and tmpGroupID
> 0)) then {
    known[dest] = POLYSIZE
    groupID[dest] = tmpGroupID
}


MAX (1 or more collections -> int)
tmpKnow = smallest of know(src)
if (groupID(src that are know()=POLYSIZE) is the same for
all such collections) then tmpGroupID = this value
else tmpGroupID = 0
if (tmpKnow = KNOWN or (tmpKnown = POLYSIZE and tmpGroupID
> 0)) then {
    known[dest] = POLYSIZE
    groupID[dest] = tmpGroupID
}
```