The Program Understanding Problem: Analysis and A Heuristic Approach

Steven Woods and Qiang Yang

Department of Computer Science
University of Waterloo,
Waterloo, Ontario, CANADA

School of Computing Science
Simon Fraser University
Burnaby, B.C., CANADA

March, 1996

Large Legacy Code is Hard to Maintain

→ *Large* legacy source code presents many challenges for software maintainers :

- Code becomes increasingly difficult to extend or debug
  - ⋆ Dispersed, local, or non-existent knowledge of code function,libraries
  - ⋆ Contains old embedded unused code
  - ⋆ Redundant code and functionality
- Inconsistent, outdated and non-existent documentation
- ... many other problems ...

A Legacy Code Fragment to Consider

```
main()
{
   // documentation describing an older version of code
   char* A, B, C;
  A = "s" + "t" + "r" + "i" + "n" + "g" + "1";
  B = "string2"
  // description of non-existing variables, omitting current
  sz = 7;
  for (int j = sz;  j>0; j--){
      C[sz-j] = B[sz-j]; }
  C[sz] = "3";
  for (int i=0; B[i]; i++){
       print(B[i]); }
  // comment referring to non-existent code
  for (int j=0; j++){
       printf("%s",C[i]); }
  // reference to outdated documentation, old library
  for (int k=0; A[k]; k++){
      outchar(A[k]); }
  }      // end of main
```

Legacy Code Maintenance

→ Maintainers attempt to alleviate these problems by :

- Rewriting all or some systems if practical
- Consolidating code functionality where possible
- Simplifying code via shared or commerical libraries

→ In order to manipulate legacy source, maintainers must:

- Possess knowledge about *domain*, *general* program plans
- *Understand* a given legacy source in this context

→ **But what does it mean to** ***Understand*** **legacy source ?**

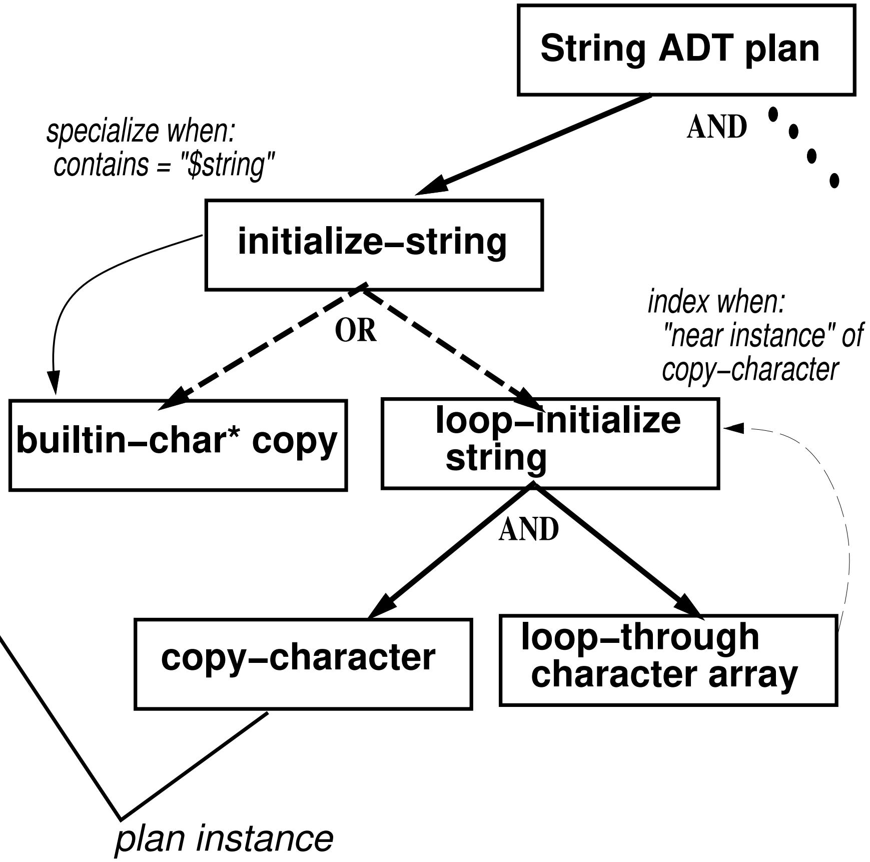
→ **How hard is it to** ***Understand*** **legacy source ?**

Understanding Legacy Code : Knowledge Mapping

Legacy Source Code

```
main()
{
  char* A, B, C;
  ...

  A = "s" + "t" + "r" + "i" + "n" + "g" +  "1";
   ...

  B = "string 2";
  ...
  sz = 7;
  for (int j = sz; j > 0; j––) {
      ....
      C[sz – j] = B[sz – j];
      ... }
  ...
  C[sz] = 3;
  ...

  for (int i=0; B[i]; i++)
       ...
     print(B[i])
       ....
  ...

  for (int j=0; C[j];j++) {
       ....
       printf("%s",C[i]);
       ... }
  ...
  for (int k=0;A[k]; k++) {
       ...
       outchar(A[k]);
       ... }
  ...}
```

Program Plan Library (excerpt)



How to Do Automatic Program Understanding ?

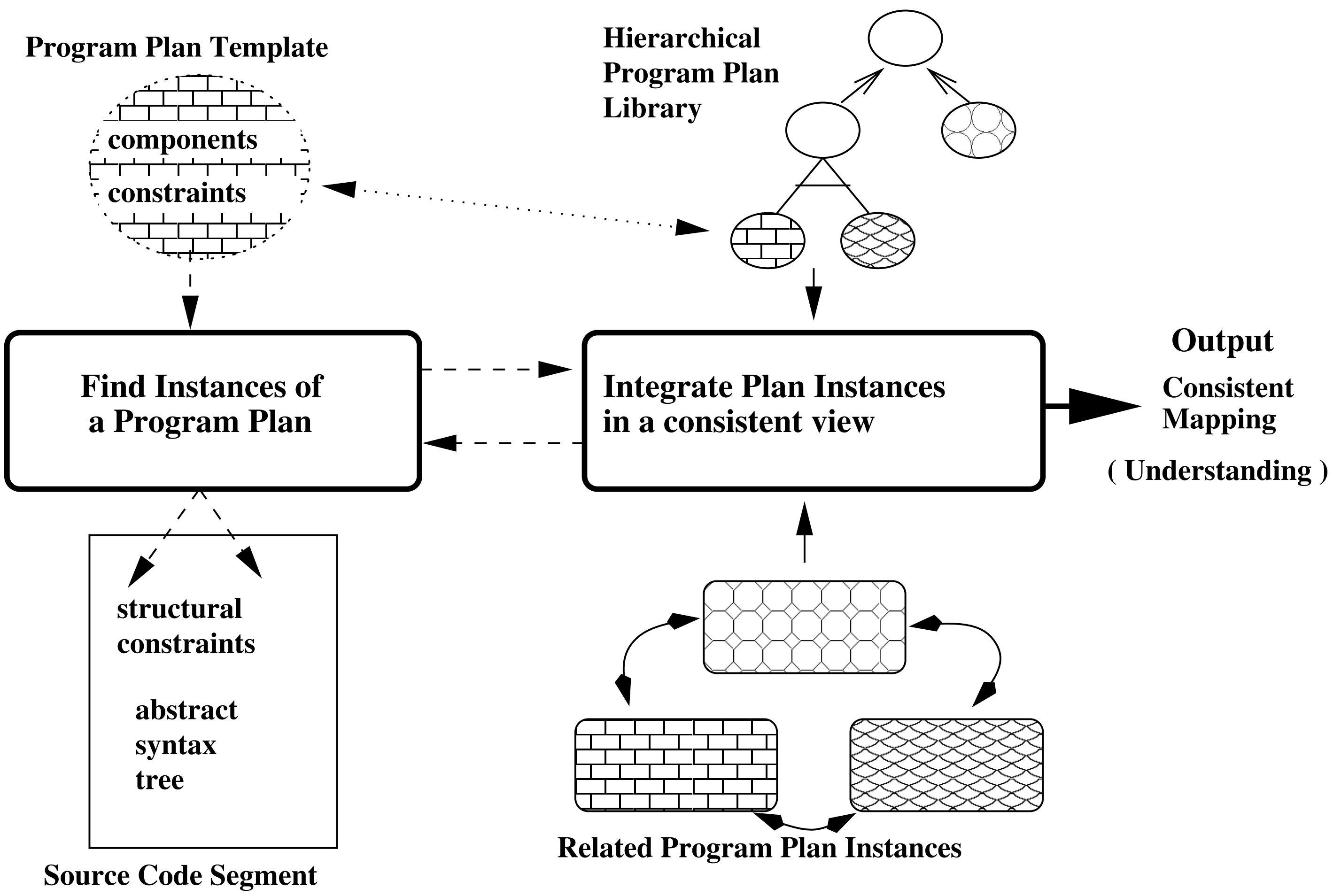
→ Directly solve the problem with a specifically designed algorithm.

- Does a polynomial algorithm exist ?

→ Generate heuristic solutions to the problem and select one.

- How do we generate these solutions ?
- How do we select amongst possible solutions ?
- Can this heuristic method scale for real/useful cases ?

Typical Program Understanding in Two Parts



Selected Past Approaches for Understanding

→ CONCEPT RECOGNIZER

- Abstracted plans consist of components and constraints
- Source is abstract syntax tree with constraint annotations

→ DECODE Index-based Recognizer extends CONCEPT RECOGNIZER

- Indices (memory keys) control candidate plan selection

→ Graph Grammar Parser

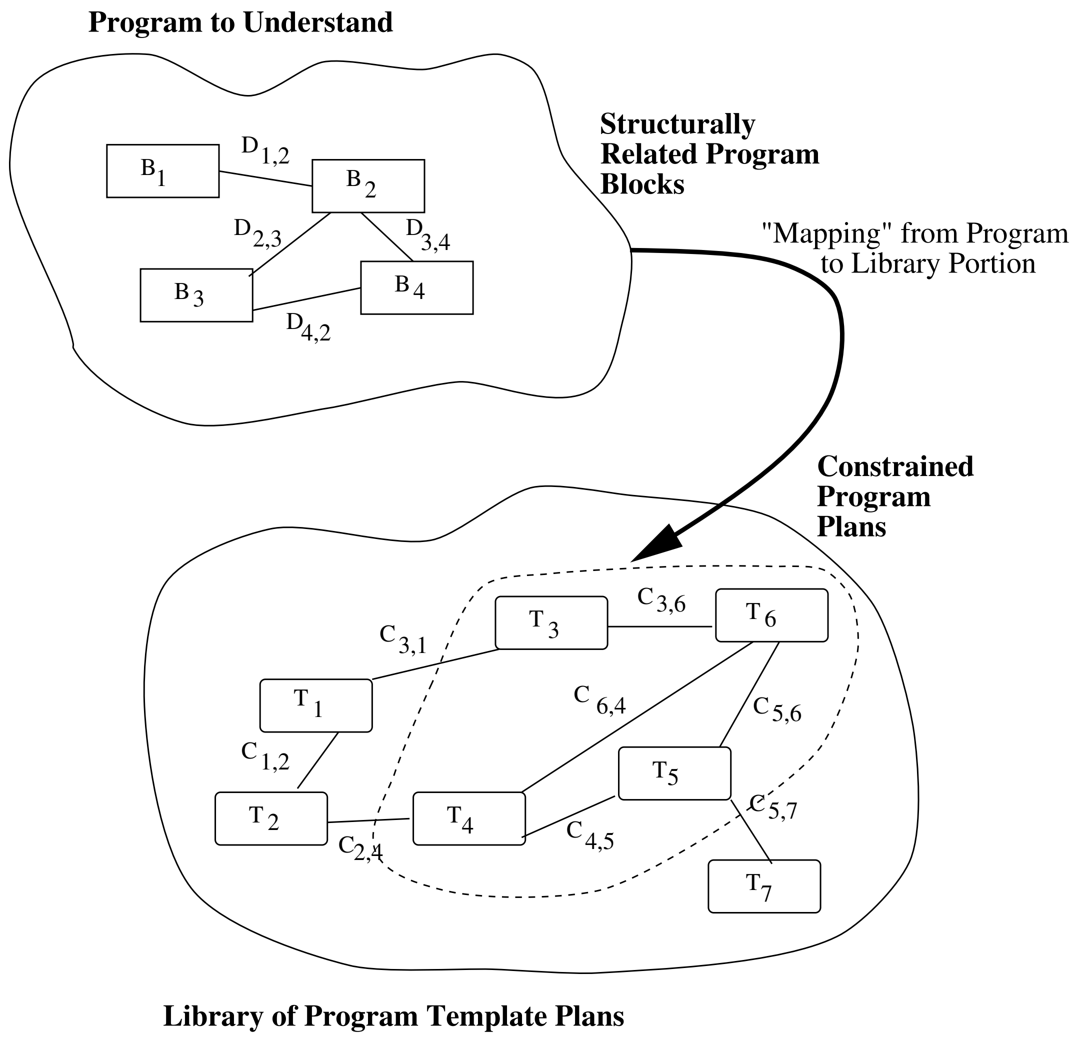
- Library components rep'd as graph grammars
- Source program rep'd as annotated flow graph
- Constrained parsing of flow graph gives explanation

Simplify Program Understanding Problem

- Want to extract minimum required problem.
- Generate simplest case including all shared features.

$\Rightarrow$ If simplified version computationally hard, then so is a more general version.

Simple Program Understanding Problem (SPU)



SPU is Representative of Past Work

→ Library: set of constraint graphs for component combination

- Abstracted, constrained, structured cliché representation

→ Source Program: annotated control and data flow graph of program blocks

- Explicit structural constraint information added to intermediate representation

→ Solution Strategy to identify mapping

- Solution must support structural constraints; satisfy component constraints

SPU is NP-hard

→ **Subgraph Isomorphism (SI)**, problem to determine if a particular graph G "contains" another graph H is NP-hard.

→ NP-hard means worst-case exponential computation, inevitable for any heuristic.

Sketch of NP-hard proof for SPU ...

→ **SI** can be transformed to Simple Program Understanding (SPU):

- Consider the library of program plans as graph G.
- Consider the source program to explain as graph H.

→ This transformation for G and H can be done in time polynomial in the combined size of the respective library and program.

Is a Simpler Problem still NP-hard ?

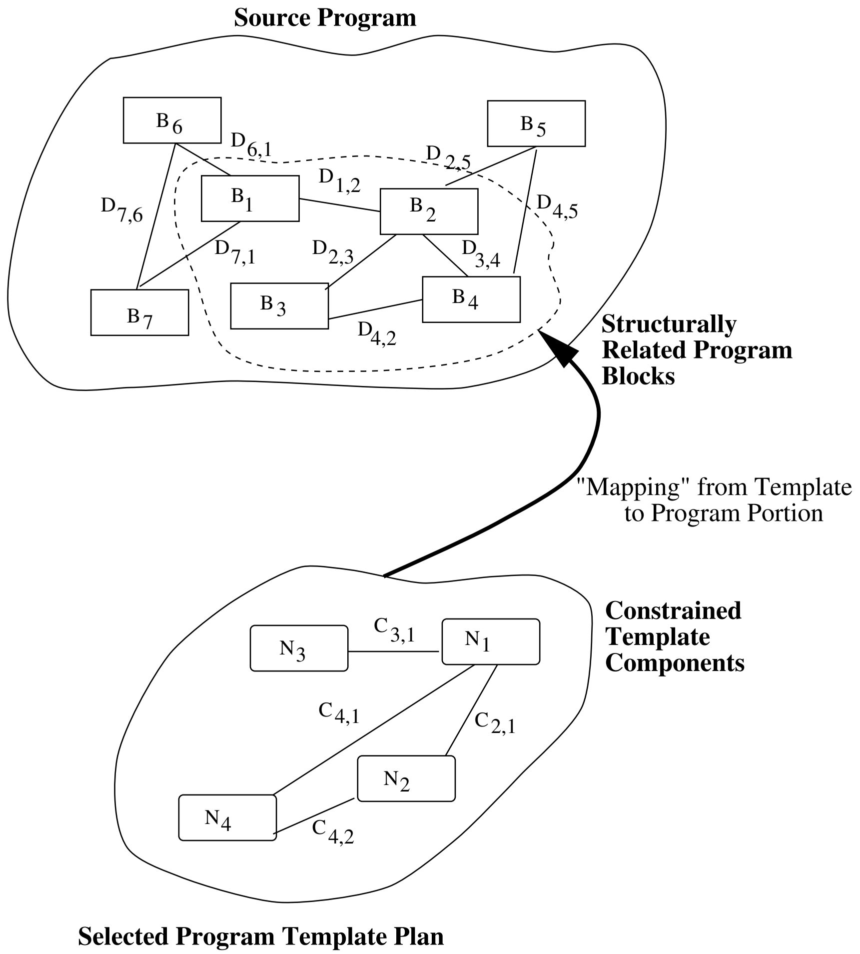
- Understanding involves explaining an entire legacy source.
- Can we explain only a segment of the source ?
- We can look for instances of a particular plan.
- Is this computationally easier ?

Simplify Template Matching Problem

- Want to extract minimum required problem.
- Generate simplest case including all shared features.

$\Rightarrow$ If simplified version computationally hard, then so is a more general version.

Simple **Program Template Matching Problem (SMAP)**



SMAP is Representative of Past Work

→ Program Plan Template (Cliché)

- Abstracted, constrained cliché components

→ Source Program: annotated control and data flow graph of program blocks

- Explicit structural constraint information added to intermediate representation

→ Solution Strategy

- Solution must satisfy template component constraints; support structural constraints

SMAP is NP-hard

- $\rightarrow$ **Subgraph-Isomorphism** is reducible to Simple Program Template Matching (SMAP).
- $\rightarrow$ The transformation is similar to SPU and is also polynomial.

So What Can We Do ?

- $\rightarrow$ Simplified problem structures resemble known framework.
- $\rightarrow$ Can this methodology help us ?

Unifying Past Approaches: Constraint Satisfaction

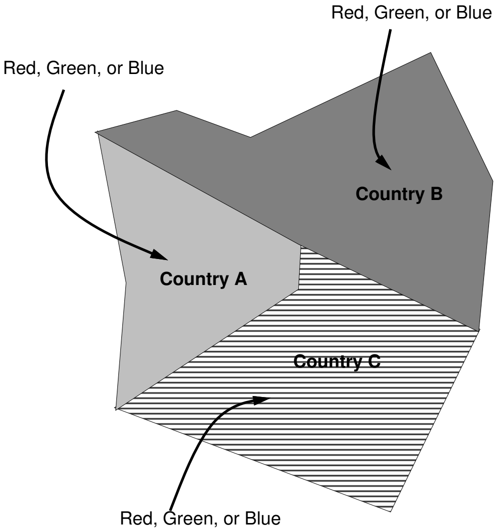


Figure 1: Map Color Example.

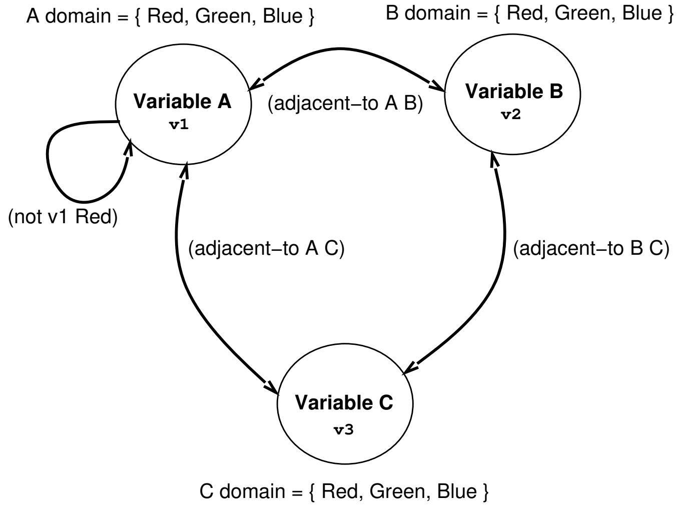


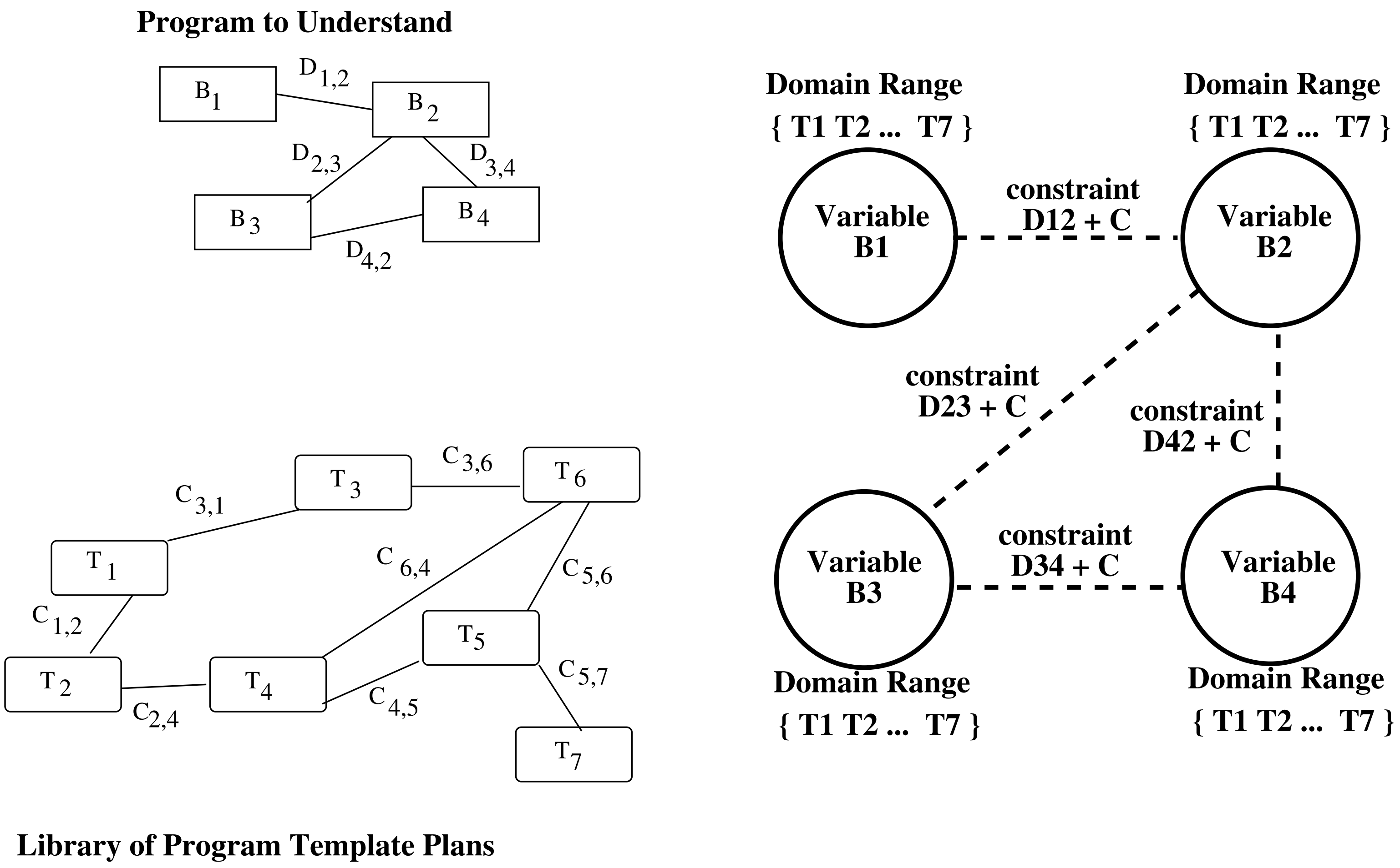
Figure 2: Map Color CSP.

→ Known methods: Local/Global Search; Constraint Propagation

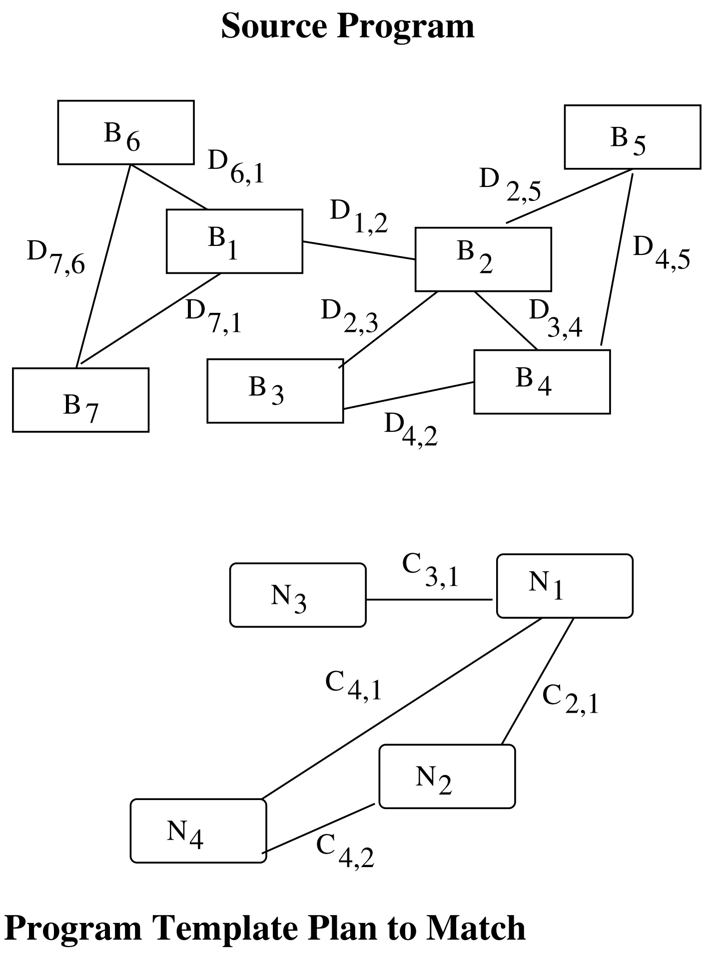
→ Increased problem knowledge ⇒ well-constrained problem

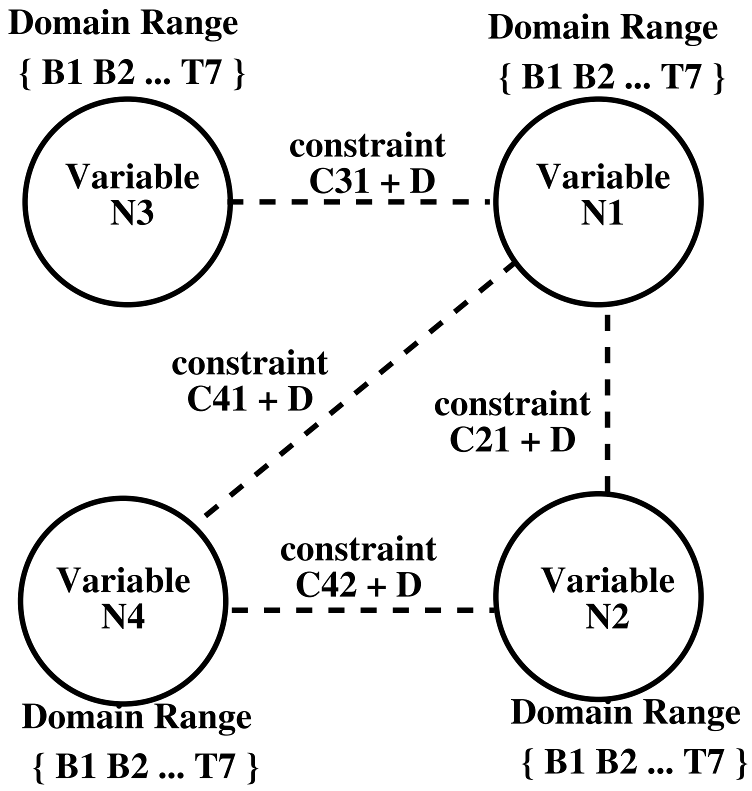
⇒ improved performance potential

SPU becomes PU-CSP



SMAP becomes MAP-CSP





Some MAP-CSP Implementation Results

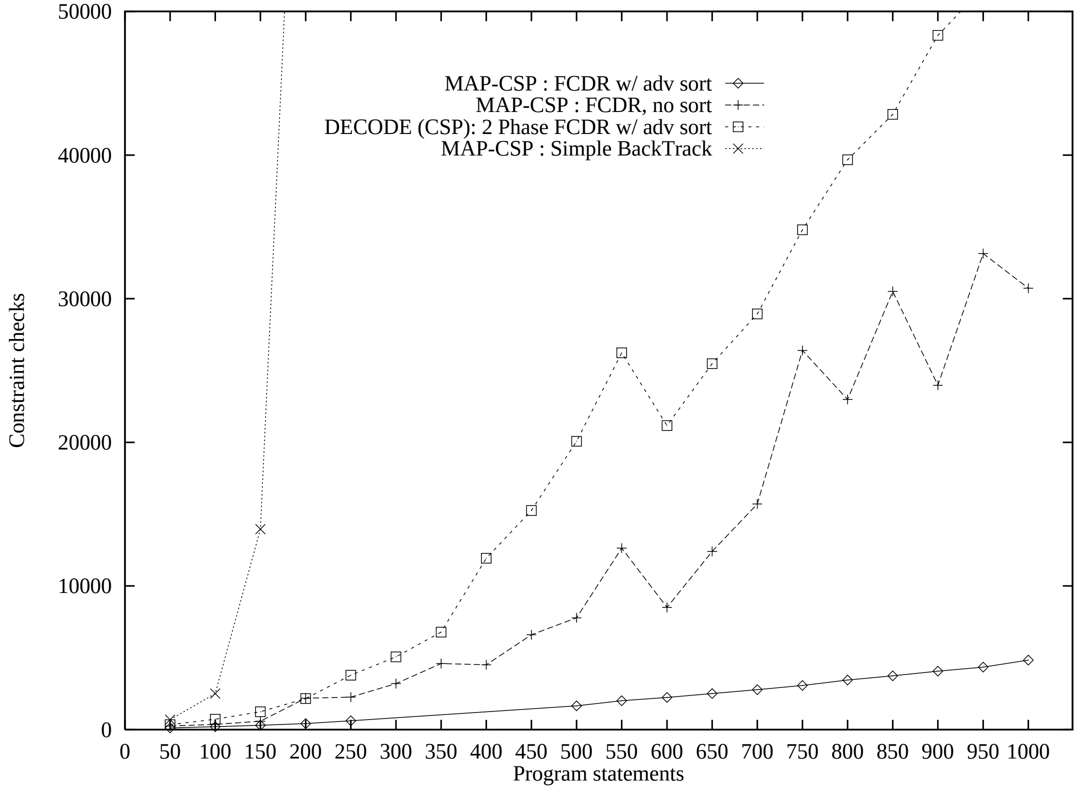


Figure 3: Matching a Template with 9 components, 20 constraints.

Conclusions

- Program Understanding is NP-hard
- Constraint Satisfaction offers a promising avenue for unifying previous heuristic approaches
- Experimental results suggest careful constraint application can be effective for SMAP/MAP-CSP in instances of $>$ 800 source lines

Ongoing Work

- Overall (integrated) understanding implementation
- Hierarchical constraint application algorithm
- Extended experiments with MAP-CSP and PU-CSP
- Looking for large, structured datasets/sources