```cpp
#include<iostream>
#include<fstream>
#include<sstream>
#include<string>
#include<iterator>
#include<stdexcept>
#include<cstdlib>
#include<cstring>
#include<cassert>
#include"debug.h"

#include "main.h"

#include "Symtab.hpp"
#include "Scanner.hpp"
#include "Algorithms.hpp"

#define DEBUG false

using namespace std;

std::ofstream* file = nullptr;

void openOutputFile() {
  if (args.output_file != nullptr) {
    try {
      file = new ofstream();
      file->exceptions(ifstream::failbit | ifstream::badbit);
      file->open(args.output_file);
    }
    catch (ios_base::failure) {
      cerr << "Error writing file \"" << args.output_file << "\".";
      debug << endl;
      exit(EXIT_FAILURE);
    }
  }
}

void closeOutputFile() {
  if (args.output_file != nullptr) {
    try {
      file->close();
    }
    catch (ios_base::failure) {
      cerr << "Error writing file \"" << args.output_file << "\".";
      debug << endl;
      exit(EXIT_FAILURE);
    }
  }
}

error_t parse_args(int argc, char *argv[]) {
  /* Set default options */
  setDefaults(args);
  /* Invoke argp parser */
  bool debug_flag = debug_enabled;
  disable_debug();
  /* Could this return an error? Do we need to check? */
  error_t result = argp_parse(&argp, argc, argv, 0, 0, &args);
  if (debug_flag) enable_debug();
  return result;
}

void readInput(Scanner& scanner) {
  if (args.input_file == nullptr) {
    debug << "Terminate with [RETURN] followed by [CTRL+D]." << endl;
    debug << "> ";
    /* Read relation from the standard input stream. */
    cin >> std::noskipws;
    std::istream_iterator<char> iter(std::cin);
    std::istream_iterator<char> end;
    /* Can this fail and if so, in what way? */
    scanner.fromString(string(iter, end));
  }
  else {
    try {
      scanner.readFile(args.input_file);
    }
    catch (ios_base::failure) {
      cerr << "Error reading file \"" << args.input_file << "\"." << endl;
      exit(EXIT_FAILURE);
    }
  }
}

void writeOutput(Symtab& symtab, TCAlgorithm& algorithm) {
  debug << "[Transitive Closure]" << endl;
  try {
    if (args.encode_output) {
      encode_mode_t mode = args.encode_mode;
      algorithm.encode(outp, mode);
      if (args.output_file != NULL) {
        algorithm.encode(debug, mode);
      }
    }
    else if (args.check_cyclic) {
      algorithm.output(symtab, debug);
      debug << endl;
      algorithm.cyclic(outp);
    }
    else {
      algorithm.output(symtab, outp);
      if (args.output_file != NULL) {
        algorithm.output(symtab, debug);
      }
    }
    /*debug << endl;*/
  }
  catch (ios_base::failure) {
    cerr << "Error writing file \"" << args.output_file << "\".";
    debug << endl;
    exit(EXIT_FAILURE);
  }
}

void parseRelation(Symtab& symtab, Scanner& scanner, TCAlgorithm& algorithm) {
  try {
    debug << "Scanning relation (pass 1)..." << endl;
    scanner.scanRelation(symtab);
    debug << "[Symbol table]" << endl;
    symtab.print(debug);
    algorithm.init(symtab.size());
    debug << "Scanning relation (pass 2)..." << endl;
    MapletRecorder mrec(algorithm);
    scanner.scanRelation(symtab, &mrec);
    debug << "[Initial Relation]" << endl;
    algorithm.output(symtab, debug);
    debug << endl;
  }
  catch (ScanException& e) {
    e.displayError(); debug << endl;
    exit(EXIT_FAILURE);
```

```cpp
  }
}

void execute(TCAlgorithm& algorithm) {
  debug << "Executing " << algorithm.name() << " algorithm..." << endl;
  algorithm.execute();
}

int main(int argc, char* argv[]) {
  if (DEBUG) enable_debug();
  Symtab symtab;
  TCAlgorithm *algorithm;
  /* Parse options */
  error_t result = parse_args(argc, argv);
  if (result != 0) { return EXIT_FAILURE; }
  openOutputFile();
  /* Read input */
  Scanner scanner;
  readInput(scanner);
  /* Parse Relation */
  algorithm = Algorithms.create(args.algorithm);
  parseRelation(symtab, scanner, *algorithm);
  /* Execute algorithm */
  execute(*algorithm);
  /* Write result */
  writeOutput(symtab, *algorithm);
  /* Report success and clean up */
  cerr << "[OK]"; debug << endl;
  closeOutputFile();
  delete algorithm;
  return EXIT_SUCCESS;
}
#ifndef MAIN_H
#define MAIN_H
#include<iostream>
#include<fstream>

#include"argp_conf.h"

#define outp (file != nullptr ? (*file) : std::cout)

extern std::ofstream* file;
#endif
#include<cstring>
#include"errno.h"
#include"debug.h"

#include "argp_conf.h"

/* Global args_t structure. */
struct args_t args;

/* Argp Setup */
const char *argp_program_version = "transcl 1.0";

const char *argp_program_bug_address = "<frank.zeyda@york.ac.uk>";

const char algorithm_option_doc[] =
  "Select the algorithm to be used. Possible values are floyd-warshall and boost,
the latter using graph algorithms of the C++ Boost Library. The default value is
floyd-warshall.";

const char encode_option_doc[] =
  "Encode output for reconstruction in Isabelle/HOL. Possible values for TYPE are
\"rel\" and \"set\", where the latter merely outputs the range set of the closure
but not the entire relation.";
```

```
const char acyclic_option_doc[] =
  "Only determine whether the relation is acyclic. This outputs either \"true\" or
\"false\".";

const char output_file_option_doc[] =
  "Output to FILE instead of the standard output.";

struct argp_option options[] = {
  {"algorithm",   'a', "NAME",  0, algorithm_option_doc },
  {"encode",      'e', "TYPE",  OPTION_ARG_OPTIONAL, encode_option_doc },
  {"cyclic",      'c', nullptr, 0, acyclic_option_doc },
  {"output-file", 'o', "FILE",  0, output_file_option_doc },
  { 0 }
};

error_t parse_opt (int key, char *arg, struct argp_state *state) {
  struct args_t *args = (struct args_t *) state->input;
  switch (key) {
    case 'a':
      try {
        args->algorithm = Algorithms.get_by_name(arg);
      }
      catch (invalid_argument& e) {
        cerr << "transcl: invalid algorithm name " << "\"" << arg << "\"";
        cerr << endl;
        argp_state_help(state, stdout, ARGP_HELP_SEE);
        return EINVAL;
      }
      break;

    case 'e':
      args->encode_output = true;
      if (arg != nullptr) {
        if (strcmp(arg, "rel") == 0) {
          args->encode_mode = relation;
        }
        else
        if (strcmp(arg, "set") == 0) {
          args->encode_mode = range_set;
        }
        else {
          cerr << "transcl: invalid encoding type " << "\"" << arg << "\"";
          cerr << endl;
          argp_state_help(state, stdout, ARGP_HELP_SEE);
          return EINVAL;
        }
      }
      break;

    case 'c':
      args->check_cyclic = true;
      break;

    case 'o':
      args->output_file = arg;
      break;

    case ARGP_KEY_INIT:
      debug << "ARGP_KEY_INIT" << endl;
      break;

    case ARGP_KEY_ARG:
      debug << "ARGP_KEY_ARG" << endl;
      if (state->arg_num == 0) {
        args->input_file = arg;
      }
      else {
        return ARGP_ERR_UNKNOWN;
      }
      break;

    case ARGP_KEY_NO_ARGS:
      debug << "ARGP_KEY_NO_ARGS" << endl;
      break;

    case ARGP_KEY_END:
      debug << "ARGP_KEY_END" << endl;
      if (args->encode_output && args->check_cyclic) {
        cerr << "options --encode and --cyclic are mutually exclusive";
        cerr << endl;
        argp_state_help(state, stdout, ARGP_HELP_SEE);
        return EINVAL;
      }
      break;

    case ARGP_KEY_SUCCESS:
      debug << "ARGP_KEY_SUCCESS" << endl;
      break;

    case ARGP_KEY_ERROR:
      debug << "ARGP_KEY_ERROR" << endl;
      break;

    case ARGP_KEY_FINI:
      debug << "ARGP_KEY_FINI" << endl;
      break;

    default:
      return ARGP_ERR_UNKNOWN;
  }
  return 0;
}

const char args_doc[] = "[INPUT_FILE]";

const char help_doc[] = "\n  The transcl command calculates and outputs the \
transitive closure of a relation, supporting various algorithms. If no \
INPUT_FILE is specified, the relation is read from the standard input. By \
default, the result is written to the standard output, unless the option -o \
is specified.\n\nOptions:";

struct argp argp = { options, parse_opt, args_doc, help_doc };

void setDefaults(args_t& args) {
  args.algorithm = floyd_warshall;
  args.encode_output = false;
  args.encode_mode = relation;
  args.check_cyclic = false;
  args.input_file = nullptr;
  args.output_file = nullptr;
}
#ifndef ARGP_CONF_H
#define ARGP_CONF_H
#include "argp.h"
#include "encode_mode.h"

#include "Algorithms.hpp"

struct args_t {
  algorithm_t algorithm;
  bool encode_output;
  encode_mode_t encode_mode;
  bool check_cyclic;
  char *input_file;
  char *output_file;
};

extern struct args_t args;

extern struct argp argp;

/* Function Prototypes */
void setDefaults(args_t& args);
#endif
#ifndef ENCODE_MODE_H
#define ENCODE_MODE_H
typedef enum { relation, range_set } encode_mode_t;
#endif
#include "debug.h"

bool debug_enabled = false;

void enable_debug() {
  debug_enabled = true;
}

void disable_debug() {
  debug_enabled = false;
}
#ifndef DEBUG_H
#define DEBUG_H
#include<iostream>

#include "NullStream.hpp"

/* Global variables */
extern bool debug_enabled;

#define debug (debug_enabled ? std::cerr : std::nil)

/* Function prototypes */
void enable_debug();
void disable_debug();
#endif
#include<iostream>
#include<sstream>
#include<string>
#include<cstring>
#include"debug.h"

#include "Scanner.hpp"

using namespace std;

Scanner::Scanner() { }

Scanner::Scanner(string filename) throw (ios_base::failure) {
  readFile(filename);
  initScan();
}

Scanner::~Scanner() {
  freeMemory();
}

void Scanner::fromString(const string& text) {
  freeMemory();
  file_size = text.size();
  file_content = new char[text.size() + 1];
  strcpy(file_content, text.c_str());
  file_content[file_size] = '\0';
}

void Scanner::readFile(string filename) throw (ios_base::failure) {
  freeMemory();
  ifstream file;
  file.exceptions(ifstream::failbit | ifstream::badbit);
  file.open(filename);
  file.seekg(0, ios::end);
  file_size = file.tellg();
  /* For homogeneity (next_char), we add an additional entry. */
  file_content = new char[file_size + 1];
  file_content[file_size] = '\0';
  file.seekg(0, ios::beg);
  file.read(file_content, file_size);
  file.close();
  initScan();
}

void Scanner::freeMemory() {
  if (file_content != nullptr) {
    delete[] file_content;
    file_content = nullptr;
    file_size = 0;
  }
}

void Scanner::initScan() {
  assert(file_content != nullptr);
  scan_index = 0;
  line = pos = 0;
  curr_valid = false;
  /* This is safe since we increase the size of file_content by one. */
  next_char = file_content[scan_index++];
  clearToken();
}

void Scanner::clearToken() {
  token[0] = '\0';
  token_index = 0;
  token_length = 0;
}

void Scanner::processLinePos() {
  if (valid()) {
    switch (current()) {
      case '\n':
        line++;

      case '\r':
        pos = 0;
        break;

      case '\t':
        pos += TAB_SIZE - (pos % TAB_SIZE);
        break;

      case '\b':
        if (pos > 0) { pos--; }
        break;

      /* Do we support the following two as well? */
      /*case '\f':*/
      /*case '\v':*/

      default:
        pos++;
    }
  }
}

char Scanner::advance() {
  processLinePos();
  if (!more()) {
    throw ScanException("Unexpected end-of-input", line, pos);
  }
  else {
    curr_char = next_char;
    next_char = file_content[scan_index++];
    curr_valid = true;
  }
  return curr_char;
}

void Scanner::consume() {
  assert(valid());
  if (token_index < MAX_TOKEN_SIZE) {
    token[token_index++] = current();
  }
  else {
    ScanException exn(line, pos);
    exn << "Maximum token lentgh of " << MAX_TOKEN_SIZE << " exceeded.";
    throw exn;
  }
}

bool Scanner::skipWS() {
  while (more()) {
    if (current() == ' ' ||
        current() == '\t' ||
        current() == '\n' ||
        current() == '\r' ||
        current() == '\f') {
      advance();
    }
    else { break; }
  }
  return more();
}

bool Scanner::skipSymbol(char c) {
  if (eof()) {
    ScanException exn(line, pos);
    exn << "End-of-input when expecting \"" << c << "\".";
    throw exn;
  }
  if (current() == c) {
    advance();
  }
  else {
    ScanException exn(line, pos);
    exn << "Expected \"" << c << "\" but \"" << current() << "\" was found.";
    throw exn;
  }
  return more();
}

bool Scanner::scanSymbol(char c) {
  if (eof()) {
    ScanException exn(line, pos);
    exn << "End-of-input when expecting \"" << c << "\".";
    throw exn;
  }
  if (current() == c) {
    consume();
    advance();
  }
  else {
    ScanException exn(line, pos);
    exn << "Expected \"" << c << "\" but \"" << current() << "\" was found.";
    throw exn;
  }
  return more();
}

bool Scanner::scanUntil(char until) {
  /* TODO: An open issue: should we skip whitespaces here? */
  /*skipWS();*/
  while (current() != until) {
    if (current() == '\"') {
      scanString();
      continue;
    }
    if (current() == '\'' && more() && next() == '\'') {
      scanHOLString();
      continue;
    }
    if (current() == ')' || current() == ']' || current() == '}') {
      ScanException exn(line, pos);
      exn << "Encountered ill-formed parenthesis: \"" << current() << "\".";
      throw exn;
    }
    consume();
    if (eof()) {
      ScanException exn(line, pos);
      exn << "Unexpected end-of-input when scanning for \"" << until << "\".";
      throw exn;
    }
    switch (current()) {
      case '(': advance(); scanUntil(')'); consume(); advance(); break;
      case '[': advance(); scanUntil(']'); consume(); advance(); break;
      case '{': advance(); scanUntil('}'); consume(); advance(); break;
      /* Are there other kinds of parentheses we need to consider? */
      default: advance();
    }
  }
  return more();
}

/* Problem: we also should consume the string quotation marks! */

bool Scanner::scanString() {
  try {
    scanSymbol('\"');
  }
  catch (ScanException& exn) {
    exn.str("");
    exn << "Error parsing string, expecting \".";
    throw exn;
  }
  try {
    while (!(current() == '\"')) {
      consume();
      advance();
    }
    scanSymbol('\"');
  }
  catch (ScanException& exn) {
    exn.str("");
    exn << "Unexpected end-of-file when parsing string.";
    throw exn;
  }
  return more();
}

bool Scanner::scanHOLString() {
  try {
    scanSymbol('\'');
    scanSymbol('\'');
  }
  catch (ScanException& exn) {
    exn.str("");
    exn << "Error parsing HOL string, expecting \"\'\'\".";
    throw exn;
  }
  try {
    while (!(current() == '\'' && more() && next() == '\'')) {
      consume();
      advance();
    }
    scanSymbol('\'');
    scanSymbol('\'');
  }
  catch (ScanException& exn) {
    exn.str("");
    exn << "Unexpected end-of-file when parsing HOL string.";
    throw exn;
  }
  return more();
}

bool Scanner::scanTerm(char until) {
  clearToken();
  skipWS();
  scanUntil(until);
  /* Remove any trailing white-space characters from the token. */
  while (token_index > 0) {
    char c = token[token_index - 1];
    if (c == ' ' || c == '\t' || c == '\n' || c == '\r' || c == '\f') {
      token_index--;
    }
    else { break; }
  }
  token[token_index] = '\0';
  token_length = token_index;
  return more();
}

void Scanner::scanRelation(Symtab& symtab, MapletRecorder *mrec) {
  initScan();
  if (mrec != nullptr) mrec->initialise();
  if (advance()) {
    skipWS();
    skipSymbol('{');
    skipWS();
    while (current() != '}') {
      skipSymbol('(');
      scanTerm(',');
      /*debug << "L-Term: \"" << token << "\"" << endl;*/
      int id1 = symtab[token];
      skipSymbol(',');
      scanTerm(')');
      /*debug << "R-Term: \"" << token << "\"" << endl;*/
      int id2 = symtab[token];
      skipSymbol(')');
      if (mrec != nullptr) mrec->record(id1-1, id2-1);
      skipWS();
      if (current() != '}') {
        skipSymbol(',');
        skipWS();
      }
    }
    /* We cannot advance() here since "}" may be the last symbol. */
    /*skipSymbol('}');*/
    if (more()) {
      skipSymbol('}');
      skipWS();
    }
    if (!eof()) {
      throw ScanException(
        "Input contains additional text after \"}\".", line, pos);
    }
    if (mrec != nullptr) mrec->finalise();
  }
  else {
    throw ScanException("Input is empty.", line, pos);
  }
}
#ifndef SCANNER_HPP
#define SCANNER_HPP
#include<fstream>
#include<cstddef>
#include<cassert>

#include "Symtab.hpp"
#include "MapletRecorder.hpp"
#include "ScanException.hpp"

#define MAX_TOKEN_SIZE 1024

/* Used to calculate line positions for error messages. */
#define TAB_SIZE 8

using namespace std;

/* Note that scan_index points to the next character to be read. Moreover,
 * scan_index points one character ahead. So, after, for instance, reading
 * the first two characters scan_index would be 3. For efficient and to
 * minimise elementary array accesses, we record both the current and next
 * character in local variables, giving us a look-ahead of one character. */
class Scanner {
protected:
  /* File content in memory */
  char *file_content = nullptr;
  streamsize file_size = 0;

  /* Scanning indices */
  int scan_index;
  int line, pos;
  /* For efficiency reasons */
  char curr_char;
  char next_char;
  bool curr_valid;

  /* Token management */
  char token[MAX_TOKEN_SIZE + 1];
  int token_index = 0;
```

```cpp
  size_t token_length = 0;

public:
  /* Constructors and Destructor */
  Scanner();
  Scanner(string filename) throw (ios_base::failure);
  ~Scanner();

  /* Inline Methods */
  inline bool more() {
    /* Note that scan_index points one character ahead. */
    return scan_index <= file_size;
  }

  inline bool eof() {
    /* Note that scan_index points one character ahead. */
    return scan_index == file_size + 1;
  }

  inline char valid() {
    return curr_valid;
  }

  inline char current() {
    assert(valid());
    return curr_char;
  }

  inline char next() {
    assert(more());
    return next_char;
  }

  /* Public Methods */
  void fromString(const string& text);
  void readFile(string filename) throw (ios_base::failure);
  void initScan();
  bool skipWS();
  bool skipSymbol(char c);
  bool scanSymbol(char c);
  bool scanUntil(char until);
  bool scanString();
  bool scanHOLString();
  bool scanTerm(char until);
  void scanRelation(Symtab& symtab, MapletRecorder *mrec = nullptr);

protected:
  /* Internal Methods */
  void clearToken();
  void processLinePos();
  char advance();
  void consume();
  void freeMemory();
};
#endif
#include<iostream>

#include "ScanException.hpp"

using namespace std;

ScanException::ScanException(int line, int pos) : line(line), pos(pos) { }

ScanException::ScanException(string msg, int line, int pos)
  : ScanException(line, pos) {
  str(msg);
```

```
}

ScanException::ScanException(const ScanException& obj)
  : ScanException(obj.line, obj.pos) {
  str(obj.str());
}

const char *ScanException::what() const noexcept {
  /* The below does not work since str() creates a temporary string object
   * that does not appear to outlive the method invocation. */
  /*return str().c_str();*/

  /* This works but how are we going to dealloclate the string object now?
   * Due to the method being consts, neither can we retain a handle to it! */
  return (new string(str()))->c_str();
}

void ScanException::displayError() {
  cerr << "Error in line " << line << ", pos " << pos << ": " << str();
}
#ifndef SCANEXCEPTION_HPP
#define SCANEXCEPTION_HPP
#include<string>
#include<exception>
#include<sstream>

using namespace std;

class ScanException : public exception, public stringstream {
protected:
  const int line, pos;

public:
  ScanException(int line, int pos);
  ScanException(string msg, int line, int pos);
  ScanException(const ScanException&);
  ~ScanException() = default;

  virtual const char *what() const noexcept;

  void displayError();
};
#endif
#include<iostream>
#include<cstdlib>
#include<string>
#include<vector>
#include<climits>
#include<cassert>
#include"debug.h"

#include"Symtab.hpp"

Symtab::Symtab() { clear(); }

void Symtab::clear() {
  root.reset();
  symbols.clear();
  counter = 1;
}

Symtab::index_t Symtab::operator[](const string& symbol) {
  TrieNode<index_t>* node = &root;
  for(char c : symbol) {
    node = (*node)[c];
  }
  if (!node->hasValue()) {
    /* Assert that an overflow will not occured. */
    assert(counter != 0);
    node->setValue(counter);
    /* Perhaps this is not the most efficient approach! */
    symbols.push_back(symbol);
    counter++;
  }
  /*debug << symbol << " = " << node->getValue() << std::endl;*/
  return node->getValue();
}

string Symtab::operator[](Symtab::index_t symid) {
  if (symid >= 1 && symid < counter) {
    return symbols[symid-1];
  }
  else {
    throw std::invalid_argument(
      "Identifier " + to_string(symid) + " is not in symbol table.");
  }
}

int Symtab::size() {
  return symbols.size();
}

void Symtab::print(ostream& os) {
  for(int i = 1; i <= size(); i++) {
    os << (*this)[i] << " = " << i << endl;
  }
}
#ifndef SYMTAB_HPP
#define SYMTAB_HPP
#include<iostream>
#include<string>
#include<vector>

#include"TrieNode.cpp"

using namespace std;

template class TrieNode<int>;

class Symtab {
public:
  /* Indices are represented by unsigned integers. */
  typedef unsigned int index_t;

protected:
  /* Root of the trie encoding for the symbol table. */
  TrieNode<index_t> root;
  /* Array used to store symbols by their index. */
  vector<string> symbols;
  /* Counter for generating the next index. */
  index_t counter;

public:
  Symtab();
  ~Symtab() = default;

  /* Remove all entries from the symbol table. */
  void clear();

  /* Obtain the identifier of a symobl; create if not present. */
  index_t operator[](const string& symbol);
  /* Obtain the symbol for a given identifier. */
  string operator[](index_t symid);

  /* Return the number of symbols in the table. */
  int size();

  /* Print the content of the symbol table. */
  void print(ostream& os);
};
#endif
#ifndef TRIENODE_CPP
#define TRIENODE_CPP
#include<cassert>

#include"TrieNode.hpp"

template <class T>
TrieNode<T>::TrieNode() {
  reset();
}

template <class T>
void TrieNode<T>::reset() {
  has_value = false;
  children.clear();
}

template <class T>
bool TrieNode<T>::hasValue() {
  return has_value;
}

template <class T>
T TrieNode<T>::getValue() {
  assert(hasValue());
  return value;
}

template <class T>
void TrieNode<T>::setValue(T new_value) {
  value = new_value;
  has_value = true;
}

template <class T>
TrieNode<T>* TrieNode<T>::operator[](char c) {
  return &children[c];
}
#endif
#ifndef TRIENODE_HPP
#define TRIENODE_HPP
#include<map>

template <class T>
class TrieNode {
/*friend class Symtab;*/

private:
  /* Does this node have a value? */
  bool has_value;
  /* Value of the node, if present. */
  T value;
  /* Child nodes of the trie object. */
  std::map<char, TrieNode> children;

public:
  TrieNode();
  ~TrieNode() = default;

  /* Reset the value and children of this node. */
  void reset();

  /* Does this node have a value? */
  bool hasValue();

  /* Get the value of this node. */
  T getValue();

  /* Set the value of this node. */
  void setValue(T new_value);

  /* Obtain child by index, create if it does not exists. */
  TrieNode<T>* operator[](char c);
};
#endif
#include<iostream>

#include "MapletRecorder.hpp"
#include "main.h"
#include "debug.h"

void MapletRecorder::initialise() {
  algorithm.clear();
  /* When producing output for Isabelle/UTP. */
  if (args.encode_output) {
    outp << algorithm.vertices() << ";";
  }
}

void MapletRecorder::record(index_t i, index_t j) {
  /*debug << "Maplet: (" << id1 << ", " << id2 << ")" << endl;*/
  algorithm.record(i, j);
  /* When producing output for Isabelle/UTP. */
  if (args.encode_output) {
    outp << i << ";";
    outp << j << ";";
  }
}

void MapletRecorder::finalise() {
  /* When producing output for Isabelle/UTP. */
  if (args.encode_output) {
    outp << endl;
  }
}
#ifndef RECORDER_HPP
#define RECORDER_HPP
#include "TCAlgorithm.hpp"

class MapletRecorder {
private:
  TCAlgorithm& algorithm;

public:
  MapletRecorder(TCAlgorithm& algorithm) : algorithm(algorithm) { };
  ~MapletRecorder() = default;
  virtual void initialise();
  virtual void record(index_t i, index_t j);
  virtual void finalise();
};
#endif
#include<cassert>
```

```
#include "TCAlgorithm.hpp"

size_t TCAlgorithm::maplets() {
  size_t count = 0;
  for (index_t i = 0; i < vertices(); i++) {
    for (index_t j = 0; j < vertices(); j++) {
      if (readout(i, j)) count++;
    }
  }
  return count;
}

void TCAlgorithm::output(Symtab symtab, std::ostream& ss) {
  ss << "{";
  bool first = true;
  for (index_t i = 0; i < vertices(); i++) {
    for (index_t j = 0; j < vertices(); j++) {
      if (readout(i, j)) {
        if (first) { first = false; }
        else {
          ss << ", ";
        }
        ss << "(" << symtab[i+1] << ", " << symtab[j+1] << ")";
      }
    }
  }
  ss << "}";
}

void TCAlgorithm::encode(std::ostream& ss, encode_mode_t mode) {
  switch (mode) {
    case relation:
      encode_relation(ss);
      break;

    case range_set:
      encode_rangeset(ss);
      break;

    default:
      assert(false);
  }
}

void TCAlgorithm::encode_relation(std::ostream& ss) {
  ss << vertices() << ";";
  for (index_t i = 0; i < vertices(); i++) {
    for (index_t j = 0; j < vertices(); j++) {
      if (readout(i, j)) {
        ss << i << ";";
        ss << j << ";";
      }
    }
  }
}

void TCAlgorithm::encode_rangeset(std::ostream& ss) {
  ss << vertices() << ";";
  for (index_t j = 0; j < vertices(); j++) {
    bool in_range = false;
    for (index_t i = 0; i < vertices(); i++) {
      in_range |= readout(i, j);
    }
    if (in_range) { ss << j << ";"; }
  }
```

```
}

void TCAlgorithm::cyclic(std::ostream& ss) {
  bool is_cyclic = false;
  for (index_t i = 0; i < vertices(); i++) {
    is_cyclic |= readout(i, i);
  }
  ss << (is_cyclic ? "true" : "false");
}
#ifndef TCALGORITHM_HPP
#define TCALGORITHM_HPP
#include<iostream>
#include<string>
#include<cstddef>

#include "Symtab.hpp"
#include "encode_mode.h"

typedef unsigned int index_t;

class TCAlgorithm {
public:
  virtual string name() = 0;
  virtual void init(size_t size) = 0;
  virtual void clear() = 0;
  virtual size_t vertices() = 0;
  virtual void record(index_t i, index_t j) = 0;
  virtual bool readout(index_t i, index_t j) = 0;
  virtual void execute() = 0;
  size_t maplets();
  void output(Symtab symtab, std::ostream& ss);
  void encode(std::ostream& ss, encode_mode_t mode);
  void cyclic(std::ostream& ss);

private:
  void encode_relation(std::ostream& ss);
  void encode_rangeset(std::ostream& ss);
};
#endif
#include<stdexcept>
#include<cassert>

#include "Algorithms.hpp"
#include "FloydWarshall.hpp"
#include "BoostAlgorithm.hpp"

class Algorithms Algorithms;

Algorithms::Algorithms() {
  algo_by_name["floyd-warshall"] = floyd_warshall;
  algo_by_name["boost"] = boost_algorithm;
}

algorithm_t Algorithms::get_by_name(string name) {
  try {
    return algo_by_name.at(name);
  }
  catch (std::out_of_range& e) {
    throw std::invalid_argument("unknown algorithm");
  }
}

TCAlgorithm* Algorithms::create(algorithm_t algorithm) {
  switch(algorithm) {
    case floyd_warshall:
      return new FloydWarshall();
    case boost_algorithm:
      return new BoostAlgorithm();

    default:
      assert(false);
  }
}
#ifndef ALGORITHMS_H
#define ALGORITHMS_H
#include<iostream>
#include<map>

#include "TCAlgorithm.hpp"

typedef enum { floyd_warshall, boost_algorithm } algorithm_t;

class Algorithms {
private:
  map<string, algorithm_t> algo_by_name;

public:
  Algorithms();
  ~Algorithms() = default;

  algorithm_t get_by_name(string);

  TCAlgorithm* create(algorithm_t);
};

extern class Algorithms Algorithms;
#endif
#include<stdexcept>

#include "FloydWarshall.hpp"

void FloydWarshall::init(size_t size) {
  if (size > ADJ_MATRIX_SIZE) {
    throw std::logic_error(
      "Number of vertices exceeds fixed adjecency matrix size.");
  }
  this->size = size;
}

void FloydWarshall::clear() {
  for (index_t i = 0; i < size; i++) {
    for (index_t j = 0; j < size; j++) {
      adj_matrix[i][j] = false;
    }
  }
}

void FloydWarshall::execute() {
  for (index_t k = 0; k < size; k++) {
    for (index_t i = 0; i < size; i++) {
      for (index_t j = 0; j < size; j++) {
        adj_matrix[i][j] |= adj_matrix[i][k] & adj_matrix[k][j];
      }
    }
  }
}
#ifndef FLOYDWARSHALL_HPP
#define FLOYDWARSHALL_HPP
#include<string>
#include<cstddef>
```

```cpp
#include "TCAlgorithm.hpp"

/* For now, the maximum size of the adjacency matrix is fixed in code. */
#define ADJ_MATRIX_SIZE 1024

typedef bool adj_matrix_t[ADJ_MATRIX_SIZE][ADJ_MATRIX_SIZE];

class FloydWarshall : public TCAlgorithm {
private:
  adj_matrix_t adj_matrix;
  size_t size;

public:
  FloydWarshall() : size(0) { };
  ~FloydWarshall() = default;

  /* Virtual Methods */
  void init(size_t size);
  void clear();
  void execute();

  /* Inline Methods */
  inline string name() { return "floyd-warshall"; }
  inline size_t vertices() { return size; }
  inline void record(index_t i, index_t j) { adj_matrix[i][j] = true; }
  inline bool readout(index_t i, index_t j) { return adj_matrix[i][j]; }
};
#endif
#include "BoostAlgorithm.hpp"

#include "boost/graph/transitive_closure.hpp"

using namespace boost;

BoostAlgorithm::BoostAlgorithm() : size(0), graph(nullptr) { }

BoostAlgorithm::~BoostAlgorithm() {
  if (graph != nullptr) delete graph;
}

void BoostAlgorithm::init(size_t size) {
  if (graph != nullptr) delete graph;
  graph = new graph_t(size);
  this->size = size;
}

void BoostAlgorithm::clear() {
  graph->clear();
}

void BoostAlgorithm::execute() {
  /* Supplying size here, even if correct, does not work...! */
  graph_t *result = new graph_t(/*size*/);
  transitive_closure(*graph, *result);
  if (graph != nullptr) delete graph;
  graph = result;
}
#ifndef BOOSTALGORITHM_HPP
#define BOOSTALGORITHM_HPP
#include<string>
#include<cstddef>

#include "TCAlgorithm.hpp"

#include "boost/graph/adjacency_list.hpp"
```

```
using namespace boost;

/* I am not entirely sure about the first two types below. Efficiency? */
typedef adjacency_list<vecS, vecS, directedS, index_t> graph_t;

class BoostAlgorithm : public TCAlgorithm {
private:
  graph_t *graph;
  size_t size;

public:
  BoostAlgorithm();
  ~BoostAlgorithm();

  /* Virtual Methods */
  void init(size_t size);
  void clear();
  void execute();

  /* Inline Methods */
  inline string name() { return "boost"; }
  inline size_t vertices() { return size; }
  inline void record(index_t i, index_t j) {
    add_edge(i, j, *graph);
  }
  inline bool readout(index_t i, index_t j) {
    return edge(i, j, *graph).second;
  }
};
#endif
#include "NullStream.hpp"

namespace std {
  NullStream nil;
}
#ifndef nullptrSTREAM_HPP
#define nullptrSTREAM_HPP
#include<ostream>
#include<streambuf>

class NullBuffer : public std::streambuf {
public:
  int overflow(int c) { return c; }
};

class NullStream : public std::ostream {
private:
  NullBuffer null_buffer;

public:
  NullStream() : std::ostream(&null_buffer) { }
};

namespace std {
  extern NullStream nil;
}
#endif
```