**PUNE INSTITUTE OF COMPUTER TECHNOLOGY**

**DEPARTMENT OF INFORMATION TECHNOLOGY**

**LABORATORY MANUAL**

**AY 2017-18,18-19,19-20**

**Semester-II**

**314456: SOFTWARE LABORATORY-V**

**TE-IT**

**TEACHING SCHEME: Practical: 4 Hrs/Week CREDITS: 02**

**EXAMINATION SCHEME: Term Work: 50 Marks**

**Practical: 50 Marks**

**:::|| Prepared By ||:::**

**Mrs. R. V. Kulkarni Mrs. J. B. Jagdale**

**Mrs. N. R. Sodha Mr. J. K. Kambale**

**314456: SOFTWARE LABORATORY-V**

**Prerequisites:**

1. Discrete Structure.

2. C/ C++ Programming.

3. Fundamentals of Data Structure and Files.

**Course Objectives:**

1. To learn the concepts of assembler to design and implement two pass assembler.

2. To study use of macros and its expansion process.

3. To understand lexical analyzer and parser and its applications in compiler design.

4. To learn the various algorithmic design paradigms.

5. To apply appropriate algorithmic strategy in problem solving.

6. To find the space and running time requirements of the algorithms.

**Course Outcomes:**

1. To design and implement two pass assembler for hypothetical machine instructions.

2. To design and implement different phases of compiler (Lexical Analyzer, Parser, Intermediate code generation)

3. To use the compile generation tools such as “Lex" and "YACC”.

4. To apply algorithmic strategies for solving various problems.

5. To compare various algorithmic strategies.

6. To analyze the solution using recurrence relation.

# INDEX OF LAB EXPERIMENTS

|  |  |  |
| --- | --- | --- |
| **LAB EXPT. NO.** | **PROBLEM STATEMENT** | **REVISED ON** |
|  | **Group A: System Programming** |  |
| **1.** | Write a program to implement Pass-I of Two-pass assembler for Symbols and Literal processing ( For hypothetical instruction set from Dhamdhere) considering following cases:  i. Forward references  ii.DS and DC statement  iii.START, EQU, LTORG, END.  iv.Error handling: symbol used but not defined, invalid instruction/register etc. | **18/12/2017** |
| **2.** | Write a program to implement Pass-II of Two-pass assembler for output of Assignment 1 (The subject teacher should provide input file for this assignment ) | **18/12/2017** |
| **3.** | Study Assignment for Macro Processor. (Consider all aspects of Macro Processor ) | **18/12/2017** |
| **4.** | Write a program to implement Lexical Analyzer for subset of C. | **18/12/2017** |
| **5.** | Write a program to implement a Recursive Descent Parser . | **18/12/2017** |
| **6.** | Write a program to implement calculator using LEX and YACC. | **18/12/2017** |
| **7.** | Write a program for Intermediate code generation using LEX &YACC for Control Flow statement ( Either While loop or Switch case) | **18/12/2017** |
|  | **Group B: Design & Analysis of Algorithms** |  |
| **1.** | Write a program to find Maximum and Minimum element in an array using Divide and Conquer strategy and verify the time complexity. | **18/12/2017** |
| **2.** | Write a program to solve optimal storage on tapes problem using Greedy approach. | **18/12/2017** |
| **3.** | Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity. | **18/12/2017** |
| **4.** | Write a program to solve the travelling salesman problem and to print the path and the cost using Dynamic Programming.(Assume number of cities ≤ 5) | **18/12/2017** |
| **5.** | Write a recursive program to find the solution of placing n queens on chessboard so that no two queens attack each other using Backtracking. | **18/12/2017** |
| **6.** | Write a program to solve the travelling salesman problem and to print the path and the cost using Branch and Bound. | **18/12/2017** |



**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Subject Coordinator Head of Department**

**Mrs. R.V.Kulkarni Dr. B. A. Sonkamble**

1. **GROUP A: System Programming**
2. **ASSIGNMENT NO: 1**
3. **TITLE: Pass I of Two pass assembler**
4. **Problem Statement:**

1. Write a program to implement Pass-I of Two-pass assembler for Symbols and Literal processing (For hypothetical instruction set from Dhamdhere) considering following cases

* Forward references
* DS and DC statement
* START, EQU, LTORG, END.
* Error handling: symbol used but not defined, invalid instruction/register etc.

1. **Objective:**

To study basic translation process of assembly language to machine language using a hypothetical instruction set..

To study Pass – I of two pass assembler in detail using variant - I.

1. **Theory:**

A language translator bridges an execution gap to machine language of computer system. An assembler is a language translator whose source language is assembly language.

Language processing activity consists of two phases, Analysis phase and synthesis phase. Analysis of source program consists of three components, Lexical rules, syntax rules and semantic rules. Lexical rules govern the formation of valid statements in source language. Semantic rules associate the formation meaning with valid statements of language. Synthesis phase is concerned with construction of target language statements, which have the same meaning as source language statements. This consists of memory allocation and code generation.

Data structures

Intermediate code

Data access

Control transfer

Figure 1.1 Overview of two-pass assembly

Analysis of source program statements may not be immediately followed by synthesis of equivalent target statements. This is due to forward references issue concerning memory requirements and organization of Language Processor (LP).

Forward reference of a program entity is a reference to the entity, which precedes its definition in the program. While processing a statement containing a forward reference, language processor does not posses all relevant information concerning referenced entity. This creates difficulties in synthesizing the equivalent target statements. Postponing the generation of target code until more information concerning the entity is available can solve this problem. This also reduces memory requirements of LP and simplifies its organization. This leads to multi-pass model of language processing.

1. **Language Processor Pass: -**

It is the processing of every statement in a source program or its equivalent representation to perform language-processing function.

1. **Assembly Language statements: -**

There are three types of statements Imperative, Declarative, Assembly directives. An imperative statement indicates an action to be performed during the execution of assembled program. Each imperative statement usually translates into one machine instruction. Declarative statement e.g. DS reserves areas of memory and associates names with them. DC constructs memory word containing constants. Assembler directives instruct the assembler to perform certain actions during assembly of a program,

e.g. START<constant> directive indicates that the first word of the target program generated by assembler should be placed at memory word with address <constant>

**Function Of Analysis And Synthesis Phase:**

1. **Analysis Phase: -**

Isolate the label operation code and operand fields of a statement.

Enter the symbol found in label field (if any) and address of next available machine word into symbol table.

Validate the mnemonic operation code by looking it up in the mnemonics table.

Determine the machine storage requirements of the statement by considering the mnemonic operation code and operand fields of the statement.

Calculate the address of the address of the first machine word following the target code generated for this statement (Location Counter Processing)

1. **Design of Pass I of a Two Pass Assembler:**

Tasks performed by the pass I of two-pass assembler are as follows:

1. **Pass I: -**

Separate the symbol, mnemonic opcode and operand fields.

Determine the storage-required foe every assembly language statement and update the location counter.

Build the symbol table and the literal table.

Construct the intermediate code for every assembly language statement.

**Data structures required for pass I:**

1. Source file containing assembly program.

2. OPTAB: A table of mnemonic op-codes and related information.

It has the following fields

**Mnemonic** : Such as ADD, END, DC

**TYPE** : IS for imperative, DL for declarative and AD for Assembler directive

**OP- code** : Operation code indicating the operation to be performed.

**Length** : Length of instruction required for Location Counter Processing

**No\_of\_operands**: The number of operand the instruction takes (Max).

Hash table Implementation of MOT to minimize the search time required for searching the instruction.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Index** | **Mnemonic** | **TYPE** | **OP-Code** | **Length** | **Link** |
| 0 | ADD | IS | 01 | 01 | -1 |
| 1 | BC | IS | 07 | 01 | -1 |
| 2 | COMP | IS | 06 | 01 | -1 |
| 3 | DIV | IS | 08 | 01 | 5 |
| 4 | EQU | AD | 03 | - | 7 |
| 5 | DC | DL | 01 | - | 6 |
| 6 | DS | DL | 02 | - | -1 |
| 7 | END | AD | 05 | - | -1 |
|  |  |  |  |  |  |

Figure 1.2: OPTAB table

Hash Function used is ASCII Value of the First letter of Mnemonic – 65. This helps in retrieving the op- code and other related information in minimum time. For Example the instruction starting with alphabet ‘A’ will be found at index location 0, ‘B’ at index 1, so on and so forth. If more instructions exist with same alphabet then the instruction is stored at empty location and the index of that instruction is stored in the link field. Thus instructions starting with alphabet ‘D’ will be stored at index locations 3,5,and 6. Those starting with E will be stored at 4 and 7 and the process continues.

1. SYMTB: The symbol table.

Fields are Symbol name, Address (LC Value). Initialize all values in the address fields to -1 and when symbol gets added when it appears in label field replace address value with current LC. The symbol if it used but not defined will have address value -1 which will be used for error detection.

|  |  |
| --- | --- |
| **Symbol** | **Address** |
| Loop | 204 |
| Next | 214 |

4. LITTAB: and POOLTAB : Literal table stores the literals used in the program and

POOLTAB stores the pointers to the literals in the current literal pool.

|  |  |
| --- | --- |
| **Literal** | **Address** |
| = ‘5’ |  |
| = ‘1’ |  |
| = ‘1’ |  |

5. Intermediate form used Variant 1 / Variant 2

Students are supposed to write the variant used by them.

*Algorithm*

*Pass1*

1 Open the source file in read mode.

2. if end of file of source file go to step 8.

3. Read the next line of the source program

4. Separate the line into words. These words could be stored in array of strings.

5. Search for first word is mnemonic opcode table, if not present it is a label , add this as a symbol in symbol table with current LC. And then search for second word in mnemonic opcode table

6. If instruction is found

case 1 : imperative statement

case 2: Declarative statement

case 3: Assembler Directive

Generate Intermediate code and write to Intermediate code file.

7. go to step 2.

8. Close source file and open intermediate code file

9. If end of file ( Intermediate code), go to step 13

10. Read next line from intermediate code file.

11. Write opcode, register code, and address of memory( to be fetched from literal or symbol table depending on the case) onto target file. This is to be done only for Imperative statement.

12 go to step 9.

13. Close all files.

14. Display symbol table, literal table and target file

Imperative statement case :

1. If opcode >= 1 && opcode <=8 ( Instruction requires register operand)

a. Set type as IS, get opcode, get register code, and make entry into symbol or literal table as the case may be. In case of symbol, used as operand, LC field is not known so LC could be -1. Perform LC processing LC++. Updating of symbol table should consider error handling.

1. if opcode is 00 ( stop) :

Set all fields of Intermediate call as 00. LC++

1. else register operand not required ( Read and Print)

Same as case 1, only register code is not required, so set it to zero. Here again update the symbol table. LC++

On similar lines we can identify the cases for declarative and assembler directive statements based on opcode.

**List of hypothetical instructions:**

* + - 1. **Instruction Assembly Remarks**

Opcode mnemonic

00 STOP stop execution

01 ADD first operand modified condition code set

02 SUB first operand modified condition code set

03 MULT first operand modified condition code set

04 MOVER register memory

05 MOVEM memory register

06 COMP sets condition code

07 BC branch on condition code

08 DIV analogous to SUB

09 READ first operand is not used.

10 PRINT first operand is not used.

1. **Sample Input & Output:**

**SAMPLE INPUT FILE SAMPLE OUTPUT FILE OF INTERMEDIATE CODE**

**Using Variant One**

START 202

MOVER AREG, =’5 202) (IS,04)(1)(L,1)

MOVEM AREG , A 203) (IS,05)(1)(S,1)

LOOP MOVER AREG, A 204) (IS,04)(1)(S,1)

MOVER CREG, B 205) (IS,04)(3)(S,3)

ADD CREG, = ‘1’ 206) (IS,01)(3)(L,2)

MOVEM CREG, B 207) (IS,05)(2)(S,3)

SUB CREG,A 208) (IS,02)(3)(S,1)

BC ANY, NEXT 209) (IS,07)(6)(S,4)

LTORG 210) (AD, 04)

211)

ADD CREG,B ’ 212) (IS,01)(3)(S,2)

BC LE LOOP 213) (IS,07)(2)(S,2)

NEXT SUB AREG, = ‘1’ 214) (IS,02)(1)(L,3)

BC LT, BACK 215) (IS.07)(1)(S,5)

STOP 216) (IS,00)

ORIGIN 219 217) (AD,03)

MULT CREG, B 219) (IS,03)(3)(S,3)

A DS 1 220) (DL,02) (C,1)

BACK EQU LOOP 221) (AD,02)

B DS 1 221) (DL,02) (C,1)

END 222) (AD,05)

SYMBOL TABLE

|  |  |  |
| --- | --- | --- |
| Index | Symbol | Address |
| 1 | A | 220 |
| 2 | LOOP | 204 |
| 3 | B | 221 |
| 4 | NEXT | 214 |
| 5 | BACK | 204 |

POOL TABLE LITERAL TABLE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LIT\_Ind |  | Index | LITERAL | ADDRESS |
| 01 |  | 1 | 5 | 210 |
| 03 |  | 2 | 1 | 211 |
|  |  | 3 | 1 | 222 |

**Instructions to the Students:**

Students are supposed to write about the organizations of the different data structures such as array, link list, etc.

Assumptions and limitations if any should be clearly mentioned.

Students are supposed to create two output files, one without errors & one with errors indicating the type of error.

At least following errors must be handled.

1. **Errors:**
2. **Forward reference (Symbol used but not defined): -**

This error occurs when some symbol is used but it is not defined into the program.

1. **Duplication of Symbol: -**

This error occurs when some symbol is declared more than once in the program.

1. **Mnemonic error:**
2. If there is invalid instruction then this error will occur.
3. **Register error: -**
4. If there is invalid register then this error will occur.
5. **Operand error: -**

This error will occur when there is an error in the operand field,

1. **FAQs**
2. Explain what is meant by pass of an assembler.
3. Explain the need for two pass assembler.
4. Explain terms such as Forward Reference and backward reference.
5. Explain various types of errors that are handled in two different passes.
6. Explain the need of Intermediate Code generation and the variants used.
7. State various tables used and their significance in the design of two pass Assembler.

**Reference:** System Programming and Operating System by Dhamdhere

1. **GROUP A: System Programming**
2. **ASSIGNMENT NO: 2**
3. **TITLE: Pass II of Two pass assembler**
4. **Problem Statement:**

1. Write a program to implement Pass-II of Two-pass assembler for output of Assignment 1 (The subject teacher should provide input file for this assignment )

1. **Objective:**

To study basic translation process of assembly language to machine language using a hypothetical instruction set..

To study Pass – II of two pass assembler in detail using variant - I.

1. **Theory:**

**Function Of Synthesis Phase:**

**Synthesis Phase:**

Obtain the machine operation code corresponding to the mnemonic operation code by searching the mnemonic table.

Obtain the address of the operand from the symbol table.

Synthesize the machine instruction or the machine form of the constant as the case may be.

1. **Design of a Two Pass Assembler:**

Tasks performed by the pass- II of two-pass assembler are as follows:

1. **Pass II: -**

Synthesize the target code by processing the intermediate code generated during

**Data Structure used by Pass II*:***

1. OPTAB: A table of mnemonic opcodes and related information.

2. SYMTAB: The symbol table

3. LITTAB: A table of literals used in the program

4. Intermediate code generated by Pass I

5. Output file containing Target code / error listing.

*Algorithm*

**List of hypothetical instructions:**

* + - 1. **Instruction Assembly Remarks**

Opcode mnemonic

00 STOP stop execution

01 ADD first operand modified condition code set

02 SUB first operand modified condition code set

03 MULT first operand modified condition code set

04 MOVER register memory

05 MOVEM memory register

06 COMP sets condition code

07 BC branch on condition code

08 DIV analogous to SUB

09 READ first operand is not used.

10 PRINT first operand is not used.

1. **Sample Input & Output:**

**SAMPLE PROGRAM FILE SAMPLE INPUT FILE TO PASS - II**

**Using Variant One**

START 202

MOVER AREG, =’5 202) (IS,04)(1)(L,1)

MOVEM AREG , A 203) (IS,05)(1)(S,1)

LOOP MOVER AREG, A 204) (IS,04)(1)(S,1)

MOVER CREG, B 205) (IS,04)(3)(S,3)

ADD CREG, = ‘1’ 206) (IS,01)(3)(L,2)

MOVEM CREG, B 207) (IS,05)(2)(S,3)

SUB CREG,A 208) (IS,02)(3)(S,1)

BC ANY, NEXT 209) (IS,07)(6)(S,4)

LTORG 210) (AD, 04)

211)

ADD CREG,B ’ 212) (IS,01)(3)(S,2)

BC LE LOOP 213) (IS,07)(2)(S,2)

NEXT SUB AREG, = ‘1’ 214) (IS,02)(1)(L,3)

BC LT, BACK 215) (IS.07)(1)(S,5)

STOP 216) (IS,00)

ORIGIN 219 217) (AD,03)

MULT CREG, B 219) (IS,03)(3)(S,3)

A DS 1 220) (DL,02) (C,1)

BACK EQU LOOP 221) (AD,02)

B DS 1 221) (DL,02) (C,1)

END 222) (AD,05)

SYMBOL TABLE

|  |  |  |
| --- | --- | --- |
| Index | Symbol | Address |
| 1 | A | 220 |
| 2 | LOOP | 204 |
| 3 | B | 221 |
| 4 | NEXT | 214 |
| 5 | BACK | 204 |

POOL TABLE LITERAL TABLE

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| LIT\_Ind |  | Index | LITERAL | ADDRESS |
| 01 |  | 1 | 5 | 210 |
| 03 |  | 2 | 1 | 211 |
|  |  | 3 | 1 | 222 |

**SAMPLE INPUT FILE SAMPLE OUTPUT FILE FOR**

**TARGET CODE**

202) (IS,04)(1)(L,1) 202) 04 1 210

203) (IS,05)(1)(S,1) 203) 05 1 220

204) (IS,04)(1)(S,1) 204) 04 1 220

205) (IS,04)(3)(S,3) 205) 04 3 221

206) (IS,01)(3)(L,2) 206) 01 3 211

207) (IS,05)(2)(S,3) 207) 05 2 221

208) (IS,02)(3)(S,1) 208) 03 3 220

209) (IS,07)(6)(S,4) 209) 07 6 214

210) (AD, 04) 210) 00 0 005

211) 211) 00 0 001

212) (IS,01)(3)(S,2) 212) 01 3 221

213) (IS,07)(2)(S,2) 213) 07 2 204

214) (IS,02)(1)(L,3) 214) 02 1 222

215) (IS.07)(1)(S,5) 215) 07 1 220

216) (IS,00) 216) 00 0 000

217) (AD,03) 217)

219) (IS,03)(3)(S,3) 219) 03 3 221

220) (DL,02) (C,1) 220)

221) (AD,02)221)

221) (DL,02) (C,1) 221)

222) (AD,05) 222) 00 0 001

**Instructions to the Students:**

Students are supposed to write about the organizations of the different data structures such as array, link list, etc.

Assumptions and limitations if any should be clearly mentioned.

Students are supposed to create two output files, one without errors & one with errors indicating the type of error.

At least following errors must be handled.

1. **FAQs**

1. Dfferentiate between varients.

2. What kind of errors you can hande in pass II

3. What is literal pool?

4. What paramater does END statement may have?

5. How does the size of the instruction affect the output of assembler?

6. Which kind of syntatic errors can be handled by Assembler?

**Reference**: System Programming and Operating System by Dhamdhere

1. **GROUP A: System Programming**
2. **ASSIGNMENT NO: 4**
3. **TITLE: Lexical Analysis**
4. **Problem Statement:**

Write a program to implement a lexical analyzer for a subset of ‘C’ language.

1. **Objective:**
2. To understand the basic principles in compilation.
3. To study lexical analysis phase of compiler.
4. **Theory:**

Compiler takes input as a source program & produces output as an equivalent sequence of machine instructions. This process consists of two-step processing of source program.

1. Analysis of source program.
2. Synthesis of target program.

**Analysis step**:

It consists of three sub steps

1. Lexical Analysis - Determine lexical constituents in source program.
2. Syntax Analysis - Determine structure of source string.
3. Semantic Analysis - Determine meaning of source string.
4. **Synthesis Step:**

It deals with memory allocation & code generation. The actions in analysis phase are uniquely defined for a given language. But synthesis step consists of many action instances where actions depend on the aspect concerning the execution environment of the compiler.

e.g. - Operating system interfaces target machine features such as instruction set, addressing modes, etc.

**Lexical analysis**:

The action of scanning the source program into proper syntactic classes is known as lexical analysis.

* 1. **Task of Lexical Analysis:**

1. To scan the program into basic elements or tokens of the language.
2. To build the Uniform symbol table (table of tokens).
3. To build the symbol & literal table.
4. To remove white spaces & comments.
5. To detect errors such as invalid identifier or constant.

**Data structures:**

1. Source program - Original Source program, which is scanned by compiler as string of characters.
2. Terminal Table - A permanent database that has entry for each terminal symbols such as arithmetic operators, keywords, punctuation characters such as ‘;’ , ‘,’etc

Fields: Name of the symbol.

1. Literal Table - This table is created during lexical analysis so as to describe all literals in the program.

Fields: Name of the literal.

4. Identifier Table - Created during lexical analysis and describes all identifiers in the program.

Fields: Name of the identifier.

5. Uniform Symbol Table - Created during lexical analysis to represent the program as a string of tokens, rather than of individual characters. Each uniform symbol contains the identification of the table to which it belongs.( IDN - Identifier table, LIT - Literal Table TRM - Terminal Symbol Table)and index within that table.

1. Buffer - One buffer or two buffer schemes to load source program part by part to reduce disk I/O.

\*[Keywords can be stored either in the terminal table or in the keyword table]

**Algorithm:**

1. Initialize line no to 1.
2. Read the source program line by line ( Buffer is line)
3. For each line separate the tokens such as
   1. identifier/ function name / keywords : Follow the transition diagram to detect this; i.e. letter followed by letter or digit. Search in keyword table for existence of keyword, otherwise it is identifier or function name.
   2. Integer Constant : Digit followed by digit
   3. All types of operators such as >, >=, ++, +, - etc
   4. Remove comments of the type /\* \*/
   5. Remove all white spaces.
4. Assign a line no and increment a line number.
5. Repeat steps 2-4 till end of file.

**Sample Input:**

A program in C language

main()

{

int i, no, sum, max ;

sum =0; max = -32767;

for ( i = 0; i< 10; i++)

{

scanf(“%d”, &no);

sum = sum + no;

if ( max > no) max = no;

}

printf(“sum = %d max = %d\n”, sum, max);

getch();

}

\*[Very Important : Do not have #include statement as part of C program over here. Otherwise you have to do pre-processor handling.]

**Sample Output:**

Keyword Table

|  |
| --- |
| for |
| if |
| int |
| float |

Identifier Table

|  |  |
| --- | --- |
| identifier | Attribute |
| Main | Function name |
| I |  |
| No |  |
| Sum |  |
| Max |  |
| scanf | Function name |
| printf | Function name |
| getch | Function name |

**Terminal Table**

|  |
| --- |
| ( |
| ) |
| { |
| } |
| + |
| - |
| = |
| & |
| , |
| ; |
| < |
| > |
| “ |
| ++ |

*Literal Table*

|  |  |
| --- | --- |
| Literal | attribute |
| 0 | Numeric constant |
| -32767 | Numeric constant |
| 10 | Numeric constant |
| “%d” | String constant |
| “sum = %d max = %d\n” | String constant |

*Uniform Symbol Table*

|  |  |  |
| --- | --- | --- |
|  | Type | Index |
| Main | IDN | 0 |
| ( | TRM | 0 |
| ) | TRM | 1 |
| { | TRM | 2 |
| Int | KEY | 0 |
| I | IDN | 1 |
| , | TRM | 8 |
| No | IDN | 2 |
| , | TRM | 8 |
| Sum | IDN | 3 |
| , | TRM | 8 |
| Max | IDN | 4 |
| ; | TRM | 9 |
| Sum | IDN | 3 |
| = | TRM | 6 |
| 0 | LIT | 0 |
| ; | TRM | 9 |
| Max | IDN | 4 |
| = | TRM | 6 |
| -32767 | LIT | 1 |
| ; | TRM | 9 |
| For | KEY | 0 |
| ( | TRM | 0 |
| I | IDN | I |
| = | TRM | 6 |
| 0 | LIT | 0 |

**Instructions:**

1. Students are supposed to draw the transition state diagrams for identifying literals, identifiers, relational operators, comments etc.
2. Students should use a buffer to load a part of source program to avoid large disk i/o.
3. Efficient search algorithms should be implemented for optimum time complexity.
4. Students should handle lexical errors such as invalid identifier or literal.
5. Any assumptions should be clearly mentioned.

**FAQs:**

1. What are the different phases of the compiler?
2. What is the role of lexical analysis in the process of compilation?
3. What are the tables/ data structures used as input and output of lexical analysis?
4. How keywords and identifiers are distinguished and processed?
5. What types of errors are generated by lexical analysis phase of compiler?

**Reference:**

1. Compiler: principle, techniques and tools: by Aho, Ullman and Sethi.

1. **GROUP A: System Programming**
2. **ASSIGNMENT NO: 5**
3. **TITLE: Recursive Descent Parser**
4. **Problem Statement:**

Study of Recursive Descent Parser.

1. **Objective:**
2. To understand the basic principles of top-down parsing.
3. To study recursive descent parsers.
4. **Theory:**

A *recursive descent parser* is a top-down parser, so called because it builds a parse tree from the top (the start symbol) down, and from left to right, using an input sentence as a target as it is scanned from left to right. (The actual tree is not constructed but is implicit in a sequence of function calls.) This type of parser was very popular for real compilers in the past, but is not as popular now. The parser is usually written entirely by hand and does not require any sophisticated tools. It is a simple and effective technique, but is not as powerful as some of the shift-reduce parsers.

This parser uses a recursive function corresponding to each grammar rule (that is, corresponding to each non-terminal symbol in the language). For simplicity one can just use the non-terminal as the name of the function. The body of each recursive function mirrors the right side of the corresponding rule. In order for this method to work, one must be able to decide which function to call based on the next input symbol.

Perhaps the hardest part of a recursive descent parser is the scanning: repeatedly fetching the next token from the scanner. It is tricky to decide when to scan, and the parser doesn't work at all if there is an extra scan or a missing scan.

1. Algorithm:

1. Apply left recursion removal method and remove left recursion in grammar if any.
2. Apply left factoring
3. Compute First Set.

Grammar

* + 1. S ->T L
    2. L -> + S | e
    3. T ->U M
    4. M ->\* T | e
    5. U ->(S) | V
    6. V -> 0 | 1 | ... | 9

1. Compute First Sets
   * 1. First(V) = { 0…9 }
     2. First(U) = First( (S) ) U First(V) = { ( } U{ (, 0…9 } = { (, 0…9 }
     3. First(M) = First( \*T ) U First(e) = = { \*, e }
     4. First(T) = First( UM ) = First( U )= { (, 0…9 }
     5. First(L) = First( +S ) U First(e) = = { +, e }
2. First(S) = First( TL ) = First( T )
3. First(S) = First( TL ) = First( T ) = { (, 0…9 }

Recursive descent parser

parse\_S( ) {

// S->TL

parse\_T(); parse\_L();

}

parse\_L( ) {

// L ->+ S

if (lookahead == “+”) {

match(“+”); parse\_S( );

}

// L

else ……..

}

parse\_T( ) {

// T->UM

parse\_U(); parse\_M();

}

parse\_L( ) {

// L->+ S

if (lookahead == “+”) {

match(“+”); parse\_S( );

}

// L

else….; \_

}

parse\_T( ) {

// T-> UM

parse\_U(); parse\_M();

}

parse\_U( ) {

if (lookahead == “(”) { // U->(S)

match(“(”); parse\_S( ); match(“)”);

}

else parse\_V( ); // U->V

}

parse\_V( ) {

if (lookahead == “0”) { // V->0

match(“0”);

} else if (lookahead == “1”) { // V->1

match(“1”);

} else

…

} else if (lookahead == “9”) { // V->9

match(“9”);

} else error( );

}

**FAQs:**

1. What is recursive descent parser?
2. How does it work?
3. What are the conditions that grammar must fulfill so that we can generate RDP for it?
4. Why is ti LL parser?

**Reference:**

1. Compiler: principle, techniques and tools: by Aho, Ullman and Sethi.

**GROUP B: Design and Analysis of Algorithms**

**ASSIGNMENT NO: 1**

**TITLE: Min-Max Problem**

**(Divide & Conquer)**

**Problem Statement:-**

Write a program to find Maximum and Minimum element in an array using Divide and Conquer strategy and verify the time complexity.

**Objective:-**

To learn:

* Need and significance of Divide & Conquer
* General method of Divide & Conquer
* Min-Max problem and its analysis

**Theory:-**

**A) Divide & Conquer(D&C) strategy:**

Suppose you have a problem which can be divided into independent sub problem to find solution, Divide & conquer is the best strategy to deal with that kind of problem

* ***Steps for D&C:***

1. Divide the problem into a number of subproblems that are smaller instances of the same problem.
2. Conquer the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. Combine the solutions to the subproblems into the solution for the original problem.

* ***Control Abstraction of D&C***

Algorithm DAndC (P)

{

if small(P) then return S(P) //termination condition

else

{

* + - 1. Divide P into smaller instances P1, P2, P3… Pk k≥1; or 1≤k≤n
      2. Apply DAndC to each of these sub problems.
      3. Return Combine (DAndC(P1), DAndC (P2), DAndC (P3)… DAndC (Pk)

}

}

**B) Algorithm structure:**

* ***Recursive***  ***reapply action to subproblem(s)***

**C) Min-Max problem:**

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

### Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

**Naïve method(straight forward method)**

In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

**Algorithm: Max-Min-Element (numbers[])**

max := numbers[1]

min := numbers[1]

for i = 2 to n do

if numbers[i] > max then

max := numbers[i]

if numbers[i] < min then

min := numbers[i]

return (max, min)

### Analysis

The number of comparison in Naive method is **2n - 2**.

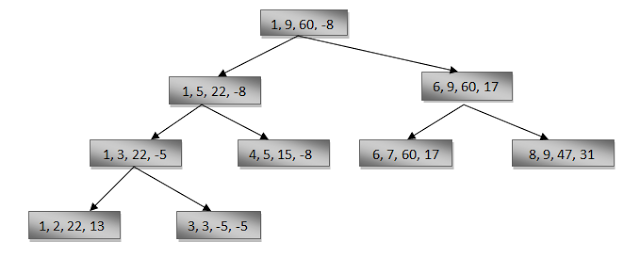
The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

**Divide & Conquer**

Use divide & conquer method to find min-max using following example.

a: [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]

    22   13   -5   -8   15  60   17   31  47



We see that the root node contains 1 and 9 as the values of i and j corresponding to the initial call to MaxMin. This execution produces two new call to MaxMin, where i and j have the values 1, 5 and 6, 9, and thus split the set into two subsets of the same size. From the tree we can immediately see that the maximum depth of recursion is four (including the first call).

**Algorithm:**

MaxMin(i, j, max, min)

// a[1:n] is a global array. Parameters i and j are integers,   // 1≤i≤j≤n. The effect is to set max and min to the largest   and smallest values in a[i:j].

{

     if (i=j) then max := min := a[i]; //Small(P)

     else if (i=j-1) then // Another case of Small(P)

          {

                if (a[i] < a[j]) then max := a[j]; min := a[i];

                else max := a[i]; min := a[j];

          }

     else

     {

           // if P is not small, divide P into sub-problems.

           // Find where to split the set.

           mid := ( i + j )/2;

           // Solve the sub-problems.

           MaxMin( i, mid, max, min );

           MaxMin( mid+1, j, max1, min1 );

           // Combine the solutions.

           if (max < max1) then max := max1;

           if (min > min1) then min := min1;

     }

}

***Complexity:***

    If T(n) represents this number, then the resulting recurrence relation is

                           0                                              n=1

T(n) =      1                                             n=2

                T(n/2) + T(n/2) + 2          n>2

     When n is a power of two, n = *2*k

-for some positive integer k, then

  T(n) = 2T(n/2) + 2

           = 2(2T(n/4) + 2) + 2

           = 4T(n/4) + 4 + 2

           .

           .

           .

           = *2*k-1 T(2) + ∑(1≤i≤k-1) *2*k

           = *2*k-1 + *2*k – 2

           = 3n/2 – 2 = O(n)

Note that 3n/2 – 2 is the best, average, worst case number of comparison when n is a power of two.

**Comparisons with Straight Forward Method*:***

        Compared with the 2n – 2 comparisons for the Straight Forward method, this is a saving of 25% in comparisons. It can be shown that no algorithm based on comparisons uses less than 3n/2 – 2 comparisons.

**Input and Output:**

**Input:** Array A having N number of elements

**Output=** Display minimum and maximum values from A

**Example:** The expected output is minimum and maximum number as -8 and 60 for following Array.

a: [1]  [2]  [3]  [4]  [5]  [6]  [7]  [8]  [9]

    22   13   -5   -8   15  60   17   31  47

**Conclusion:**

Min max problem is studied and implemented using Divide & Conquer method

**FAQ :**

1. Explain Divide & Conquer strategy, its need and characteristics.
2. Write Algorithm to solve min max problem using divide & Conquer.
3. Analyze the recursive solution for Min-max problem.
4. Is there any another algorithmic strategy to solve Min-max problem? If yes , explain the same in brief.

**GROUP B: Design & Analysis of Algorithms**

**ASSIGNMENT NO: 2**

**TITLE: Optimal Storage on tape**

**(Greedy)**

**Problem Statement:-**

Write a program to solve optimal storage on tapes problem using Greedy approach.

**Objective:-**

To learn:

* Need and significance of Greedy Algorithm
* General method of Greedy
* Optimal storage on tape algorithm and its analysis

**Theory:-**

**Greedy Algorithm:**

Suppose that a problem can be solved by a sequence of decisions. The greedy method has that each decision is locally optimal. These locally optimal solutions will finally add up to a globally optimal solution.

**General Method**

* Most problems have n inputs
* Solution contains a subset of inputs that satisfies a given constraint
* Feasible solution: Any subset that satisfies the constraint
* Need to find a feasible solution that maximizes or minimizes a given objective function – optimal solution

**Control Abstraction**

Algorithm Greedy(a , n)

// Algorithm takes as input an array a of n elements

{

Solution := {} // Initially empty

For I = 1 to n do

{

x := select(a); // Select an input from a and remove it from further consideration

If Feasible(Solution , x) then

Solution := Union(Solution , x);

}

Return Solution

}

**Example Problems**

* MST using Prim’s & Krushkal’s
* Fractional knapsack problem
* Job scheduling
* Dijkstra’s shortest path algorithm
* Optimal storage on tape

**Optimal storage on tape**

* There are n programs that are to be stored on a computer tape of length L. Associated with each program i is a length Li.
* Assume the tape is initially positioned at the front. If the programs are stored in the order I = i1, i2, …, in, the time tj needed to retrieve program ij

tj =

If all programs are retrieved equally often, then the mean retrieval time

(MRT) **=**

This problem fits the ordering paradigm. Minimizing the MRT is equivalent to minimizing

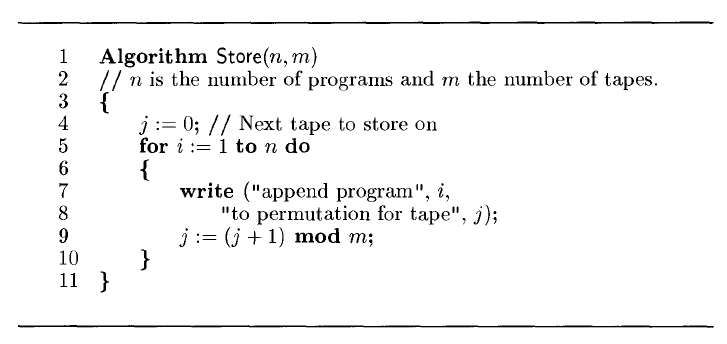
d(I) **=**

**Using Brute Force Approach:**

**Let n = 3, (L1,L2,L3) = (5,10,3). 6 possible orderings. The optimal is 3,1,2**

|  |  |
| --- | --- |
| **Ordering I** | **d(I)** |
| 1,2,3 | 5+5+10+5+10+3 = 38 |
| 1,3,2 | 5+5+3+5+3+10 = 31 |
| 2,1,3 | 10+10+5+10+5+3 = 43 |
| 2,3,1 | 10+10+3+10+3+5 = 41 |
| 3,1,2 | 3+3+5+3+5+10 = 29 |
| 3,2,1, | 3+3+10+3+10+5 = 34 |

**Using Greedy strategy**



**Analysis:**

* **Line no. 5 of algorithm requires n amount of time to execute**
* **Time required is O(n)**

**Example**

We want to store files of lengths (in MB) {12,34,56,24,11,34,34,45} on **three** tapes. How should we store them on the three tapes so that the mean retrievaltime is minimized?

**SOLUTION: STORE FILES BY NONDECREASING LENGTH**

First sort the files in increasing order of length. For this we can use heapsort, meregesort or quicksort algorithms.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 11 | 12 | 24 | 34 | 34 | 34 | 45 | 56 |

Now distribute the files:

First element 11

Tape 11

1

Tape

2

Tape

3

Second element 12

Tape 11

1

Tape 12

2

Tape

3

Third element 24

Tape 11

1

Tape 12

2

Tape 24

3

Fourth element 34

Tape 11 34

1

Tape 12

2

Tape 24

3

Fifth element 34

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Tape | 11 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 12 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 24 |  |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Sixth element 34 | | | |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 11 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 12 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 24 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Seventh element 45 | | | |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 11 | 34 |  | 45 |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 12 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 24 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Eigth element 56 | | | |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 11 | 34 |  | 45 |  |  |  |  |  |  |  |  |  |  |
| 1 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 12 | 34 |  | 56 |  |  |  |  |  |  |  |  |  |  |
| 2 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Tape | 24 | 34 |  |  |  |  |  |  |  |  |  |  |  |  |
| 3 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

**Input and Output:**

**Input= N Files with file names & sizes, M tapes**

**Output= store N files in M tapes in optimal way ( mean access time is minimum).**

**Conclusion:**

The greedy strategy for optimal storage on tape is studied and implemented.

**FAQ :**

1. Explain Greedy strategy.
2. Explain need, control abstraction and characteristics of Greedy strategy.
3. Explain following with example.
   1. Feasible solution
   2. Optimal solution
   3. Subset paradigm
   4. Ordering paradigm
4. Explain time analysis of optimal storage on tape.

**GROUP B: Design and Analysis of Algorithms**

**ASSIGNMENT NO: 3**

**TITLE: Bellman- Ford Algorithm**

**(Dynamic Programming)**

**Problem Statement:-**

Write a program to implement Bellman-Ford Algorithm using Dynamic Programming and verify the time complexity.

**Objective:-**

To learn:

* Need and significance of Dynamic programming
* General method of dynamic programming
* Bellman– Ford algorithm to and its analysis

**Theory:-**

**Dynamic Programming Strategy**

Suppose you have a problem which can be divided into independent sub problem to find solution, Divide & conquer is the best strategy to deal with that kind of problem Dynamic Programming is an algorithm design technique for **optimization problems**: often minimizing or maximizing.

**Principle of optimality**

The principle of optimality states that an optimal sequences of decisions has the property that whatever the initial state and decision are the remaining decision must constitute optimal decision sequences with regards to the stage resulting from the first decision.

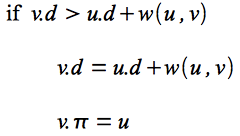
**Shortest Path Problem**

Find the shortest paths from a directed acyclic weighted graph G(V,E) starting from source to all reachable vertices. Bellman ford works with negative weighted edges and gives error if negative cycle is found in graph.

**Generic Algorithm**

The single source shortest path algorithms use the predecessor π and distance *d* fields for each vertex. The optimal solution will have *v.d* = δ(*s*,*v*) for all *v* ∈ *V*.

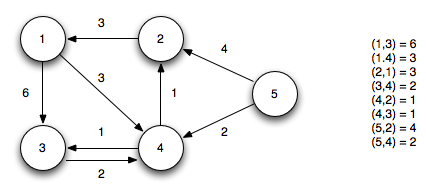
The solutions utilize the concept of ***edge relaxation*** which is a test to determine whether going through edge (*u*,*v*) reduces the distance to *v* and if so update *v.*π and *v.d*. This is accomplished using the condition



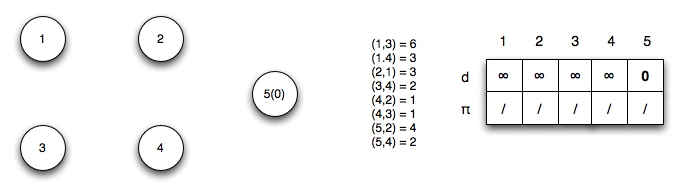
**Bellman-Ford Algorithm example**

**Example**

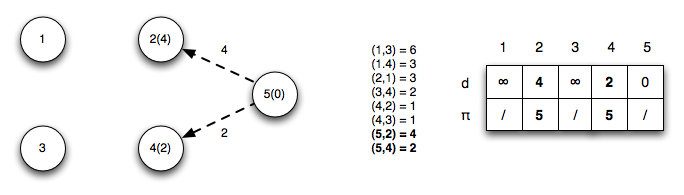
Given the following directed graph



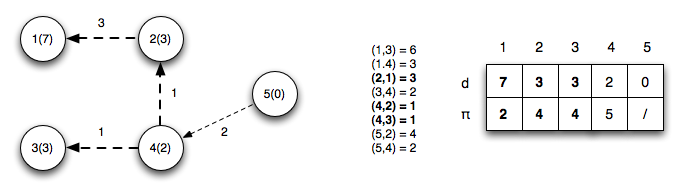
Using vertex 5 as the source (setting its distance to 0), we initialize all the other distances to ∞.



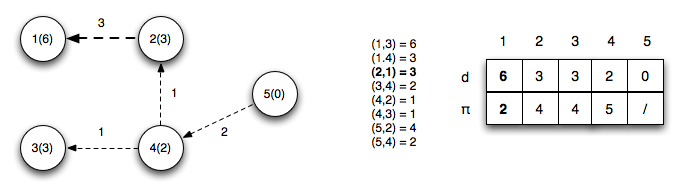
*Iteration 1*: Edges (*u*5,*u*2) and (*u*5,*u*4) relax updating the distances to 2 and 4



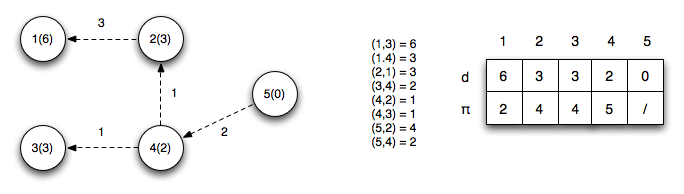
*Iteration 2*: Edges (*u*2,*u*1), (*u*4,*u*2) and (*u*4,*u*3) relax updating the distances to 1, 2, and 4 respectively. Note edge (*u*4,*u*2) finds a shorter path to vertex 2 by going through vertex 4



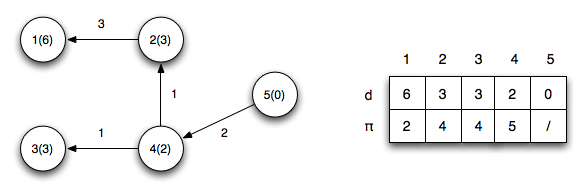
*Iteration 3*: Edge (*u*2,*u*1) relaxes (since a shorter path to vertex 2 was found in the previous iteration) updating the distance to 1



*Iteration 4*: No edges relax



The final shortest paths from vertex 5 with corresponding distances is



*Negative cycle checks*: We now check the relaxation condition one additional time for each edge. If any of the checks pass then there exists a negative weight cycle in the graph.

*v*3*.d* > *u*1*.d* + *w*(1,3) ⇒ 4 ≯ 6 + 6 = 12 ✓

*v*4*.d* > *u*1*.d* + *w*(1,4) ⇒ 2 ≯ 6 + 3 = 9 ✓

*v*1*.d* > *u*2*.d* + *w*(2,1) ⇒ 6 ≯ 3 + 3 = 6 ✓

*v*4*.d* > *u*3*.d* + *w*(3,4) ⇒ 2 ≯ 3 + 2 = 5 ✓

*v*2*.d* > *u*4*.d* + *w*(4,2) ⇒ 3 ≯ 2 + 1 = 3 ✓

*v*3*.d* > *u*4*.d* + *w*(4,3) ⇒ 3 ≯ 2 + 1 = 3 ✓

*v*2*.d* > *u*5*.d* + *w*(5,2) ⇒ 3 ≯ 0 + 4 = 4 ✓

*v*4*.d* > *u*5*.d* + *w*(5,4) ⇒ 2 ≯ 0 + 2 = 2 ✓

The path to any reachable vertex can be found by starting at the vertex and following the π's back to the source. For example, starting at vertex 1, *u*1.π = 2, *u*2.π = 4, *u*4.π = 5 ⇒ the shortest path to vertex 1 is {5,4,2,1}.

**Algorithm:**

The *Bellman-Ford algorithm* uses relaxation to find single source shortest paths on directed graphs that may contain *negative weight edges*. The algorithm will also detect if there are any *negative weight cycles* (such that there is no solution).

**BELLMAN-FORD(G,w,s)**

1. INITIALIZE-SINGLE-SOURCE(G,s)

2. for i = 1 to |G.V|-1

3. for each edge (u,v) ∈ G.E

4. RELAX(u,v,w)

5. for each edge (u,v) ∈ G.E

6. if v.d > u.d + w(u,v)

7. return FALSE

8. return TRUE

**INITIALIZE-SINGLE-SOURCE(G,s)**

1. for each vertex v ∈ G.V

2. v.d = ∞

3. v.pi = NIL

4. s.d = 0

**RELAX(u,v,w)**

1. if v.d > u.d + w(u,v)

2. v.d = u.d + w(u,v)

3. v.pi = u

***Complexity:***

Basically the algorithm works as follows:

1. Initialize *d*'s, π's, and set *s.d* = 0 ⇒ O(*V*)
2. Loop |*V*|-1 times through all edges checking the relaxation condition to compute minimum distances ⇒ (|*V*|-1) O(*E*) = O(*VE*)
3. Loop through all edges checking for negative weight cycles which occurs if any of the relaxation conditions fail ⇒ O(*E*)

The run time of the Bellman-Ford algorithm is O(*V* + *VE* + *E*) = O(*VE*).

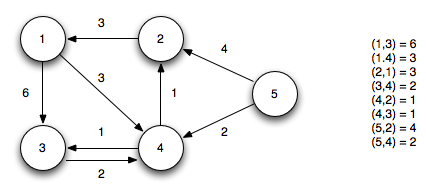
**Input and Output:**

**Input:** Graph G=(V,E) and a source vertex src

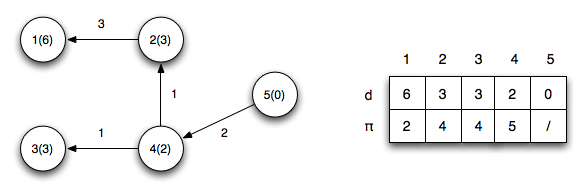
**Output=** Shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.

**Example:**

**Input:**



**Output:**



**Conclusion:**

Bellman Ford algorithm to find single source shortest path is studied and implemented using Dynamic programming method

**FAQ :**

1. Explain Dynamic programming, its need and characteristics.
2. Explain optimal sub structure and overlapping sub problems
3. Compare Dynamic programming with Greedy technique and Divide & conquer strategy
4. Write Algorithm to find single source shortest path using Bellman-Ford algorithm. Compare bellman ford with Dijkstra’s algorithm.
5. Analyze Bellman-Ford algorithm.

**GROUP B: Design & Analysis of Algorithms**

**ASSIGNMENT NO: 4**

**TITLE: Travelling Salesman Problem**

**(Dynamic Programming)**

**Problem Statement:-**

Write a program to solve the travelling salesman problem and to print the path and the cost using Branch and Bound.

**Objective:-**

To understand and implement Least Cost Branch and Bound algorithm for solving Travelling salesperson problem and study Branch and Bound strategy.

**Theory:-**

**Travelling salesperson problem:**

A traveling salesman is getting ready for a big sales tour. Starting at his hometown, suitcase in hand, he will conduct a journey in which each of his target cities is visited exactly once before he returns home. Given the pairwise costs/distances between cities, what is the best order in which to visit them, so as to minimize the overall cost/distance traveled?

Let G = *(V, E)* be a directed graph defining an instance of the traveling salesperson problem. Let *cij* be the cost of edge <i, j>, *cij* = ∞ if <i, j> does not belong to *E* and let | V| = *n.* Without loss of generality, we may assume that every tour starts and ends at vertex 1. So, the solution space *S* is given by *S* = {l, π, l | π is a permutation of (2, 3, ... , n)}. |*S*| = (n-1)! The size of *S* may be reduced by restricting *S* so that (1, i1, i2,... , in-1, 1) Є *S* iff <ij , ij+1 > Є E, 0 ≤ j ≤ n - 1, i0 = in= 1. A tour of G is a directed cycle that includes every vertex in V. The cost of a tour is the sum of the cost of the edges on the tour. The traveling salesperson problem is to find a tour of minimum cost.

**Dynamic Programming Strategy**:

Without loss of generality, regard a tour to be a simple path that starts and ends at vertex 1. Every tour consists of an edge < 1, k > for some k Є V - {1} and a path from vertex

k to vertex 1. The path from vertex k to vertex 1 goes through each vertex in V - { 1, k} exactly once. It is easy to see that if the tour is optimal then the path from k to 1 must be a shortest k to 1 path going through all vertices in V - { 1, k }. Hence, the principle of optimality holds. Let g(i, S) be the length of a shortest path starting at vertex i, going through all vertices in S and terminating at vertex 1. g( 1, V - { 1 } ) is the length of an optimal salesperson tour. From the principal of optimality it follows that:

g(l, V- {1}) = min 2≤ k ≤ n {c1k + g(k, V- {l,k})} ---------------------------(1)

Generalizing eq.(1) we obtain (for i not belonging to S):

g(i, S) = min j Є S {cij + g(j, S - {j})} ---------------------------(2)

Eq.(1) may be solved for g(l, V - { 1}) if we know g(k, V - { 1, k }) for all choices of k. The g values may be obtained by using eq.(2). Clearly, g(i, Φ) = ci1, 1 ≤ i ≤ n. Hence, we may use eq.(2) to obtain g(i, S) for all S of size 1. Then we can obtain g(i, S) for S with |S| = 2 etc. When |S| < n - 1, the values of i and S for which g (i, S) is needed are such that i ≠1; 1 and i do not belong to S.

**Algorithm for TSP using DP:**

**g(1, {1}) = 0**

**for s = 2 to n do**

**for all subsets S of size s and containing 1:**

**g(1, S) = ∞**

**for all** j Є S**; j ≠ 1 do**

g(i, S) = min j Є S {cij + g(j, S - {j})}

**return min** g(i, S)

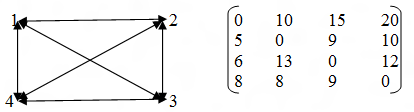
**Analysis:**

The brute-force approach is to evaluate every possible tour and return the best one. Since there are (n -1)! possibilities of visiting the n points, which results in a runtime of O(n!).

**By dynamic programming:**

There are at most 2n \* n subproblems, and each one takes linear time to solve. The total running time is therefore O(n22n).

**Example:**



g(2, Φ)= c21 = 5; g(3, Φ)= c31 = 6 and g(4, Φ)= C41 = 8

g(2, { 3}) = c23 + g(3, Φ) = 15; g(2, {4}) = 18

g(3, {2}) = 18; g(3,{4}) = 20

g(4, {2}) = 13; g(4, {3}) = 15

g(2, {3, 4}) = min{c23 + g(3, {4}), c24 + g(4, {3})} = 25

g(3, {2, 4}) = min{c32 + g(2, { 4}), c34 + g(4, {2})} = 25

g(4, {2, 3}) = min{c42 + g(2, {3}), c43 + g(3, {2})} = 23

Finally,

g(1,{2,3,4}) = min{c12 +g(2,{3,4}),c13 +g(3,{2,4}),c14 +g(4,{2,3})}

= min{ 35, 40, 43}

= 35

J(1,{2,3,4}) =2

J(2,{3,4}) =4

J(4,{3}) =3

Final optimal tour: 1-2-4-3-1 and Cost of tour= 35

**Input and Output:**

**Input= Cost matrix TSP graph**

**Output= Minimum cost of tour and its path.**

**Conclusion:**

The Dynamic programming strategy for TSP is studied and implemented.

**FAQ :**

1. Explain Dynamic programming strategy.
2. Explain difference between Dynamic programming and Greedy method.
3. Explain TSP.
4. Explain principle of optimality.

**GROUP B: Design and Analysis of Algorithms**

**ASSIGNMENT NO: 5**

**TITLE: N Queens’ Problem**

**(Backtracking)**

**Problem Statement:-**

Write a recursive program to find the solution of placing n queens on chess board so that no two queens attack each other using backtracking.

**Objective:-**

To learn:

* Need and significance of Backtracking
* General method of Backtracking
* Recursive and Iterative algorithms
* 8-queens’ problem and its analysis
* Bounding function for 8-queens’ problem
* Recursive backtracking algorithm for 8-queens’ problem

**Theory:-**

**A) Backtracking strategy:**

Suppose you have to make a series of *decisions,* among various *choices,* where

* *You don’t have enough information to know what to choose*
* *Each decision leads to a new set of choices*
* *Some sequence of choices (possibly more than one) may be a solution to your problem*

Backtracking is a methodical way of trying out various sequences of decisions, until you find one that “works”.

General steps followed in Backtracking:

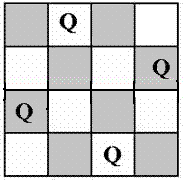
* Construct the state space tree:
  + *Root represents an initial state*
  + *Nodes reflect specific choices made for a solution’s components.*
    - Promising and nonpromising nodes
    - leaves
* Explore the state space tree using depth-first search
* “Prune” non-promising nodes
  + *dfs stops exploring subtree rooted at nodes leading to no solutions and...*
  + *“backtracks” to its parent node*

**B) Algorithm structure:**

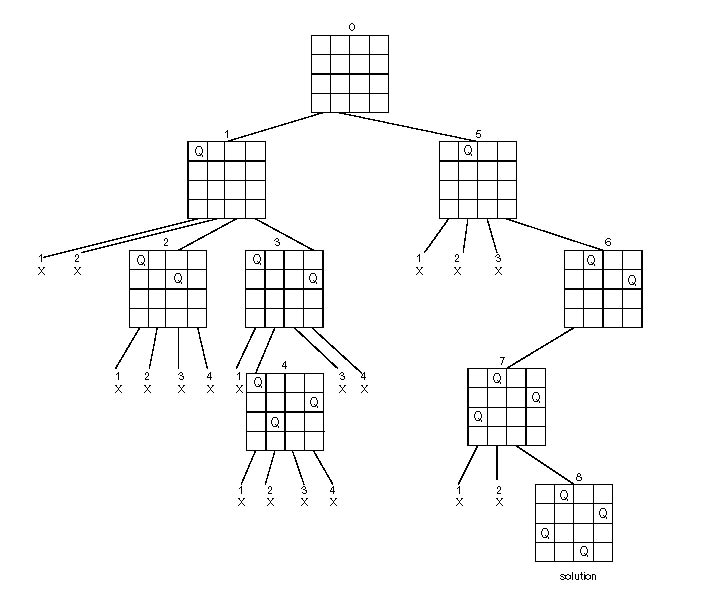
* ***Iterative*** Þ ***execute action in loop***
* ***Recursive*** Þ ***reapply action to subproblem(s)***

**C) N Queens’ problem:**

The N Queen is the problem of placing N chess queens on an N×N chessboard so that no two queens attack each other. For example, following is a solution for 4 Queen problem.



State space tree for 4 Queens’ problem:



**Algorithm:**

**Algorithm Place(k,i)**

//Returns true if a queen can be placed in kth row & ith column.Otherwise it returns false.

//x[i] is a global array whose first (k-1)values have been set.

//Abs(r) returns the absolute value of r.

{

for j:=1 to k-1 do

if ((x[j]=i)// Two in the same row

or (Abs(x[j]-i) = Abs(j-k))) // Two in the same diagonal

then return false;

return true;

}

**Algorithm NQueen(k,n)**

// Using backtracking, this procedure prints all possible placements of n queen on an

// n x n chessboard so that they are nonattacking.

{

for i:=1 to n do

{

if Place(k,i) then

{

x[k]:=i;

if (k=n) then write (x[1 : n]);

else NQueen(k+1 ,n);

}

}

}

**Analysis:**

The backtracking algorithm is more effective over the brute force approach. For 8 \* 8 chess board, there are 64C8 ways to place 8 queens or 4.4 billion 8 tuples to examine. By allowing only placements of queens on distinct rows and columns, we require to examine at most 8! tuples or 40,320 tuples.

**Input and Output:**

**Input :** N=no.of queens to be placed on N x N chessboard.

**Output=**all possible solutions to N Queens’ problem

**Example:** The expected output is a binary matrix which has 1s for the blocks where queens are placed. For example following is the output matrix for one of the solution of 4 queens’ solution.

{ 0, 1, 0, 0} OR (-- Q -- --)

{ 0, 0, 0, 1} OR (-- -- -- Q)

{ 1, 0, 0, 0} OR (Q -- -- --)

{ 0, 0, 1, 0} OR (-- -- Q --)

**Graphical output is not expected, if done most welcome!**

**Conclusion:**

N Queens’ problem is studied and implemented using recursive backtracking strategy.

**FAQ :**

1. Explain backtracking strategy, its need and characteristics.
2. Explain the significance of bounding function in backtracking.
3. Explain terms: a) Solution space b) State space c) Answer state d) Static trees e) Dynamic trees f) Live nodes g) Bounding function h) Explicit constraint i) Implicit constraint
4. What are bounding conditions used in solving N Queens’ problem?
5. Analyze the recursive backtracking solution for N Queens’ problem.
6. Is there any another algorithmic strategy to solve N Queens’ problem? If yes , explain the same in brief.

**GROUP B: Design & Analysis of Algorithms**

**ASSIGNMENT NO: 6**

**TITLE: Travelling Salesman Problem**

**(Branch and Bound)**

**Problem Statement:-**

Write a program to solve the travelling salesman problem and to print the path and the cost using Branch and Bound.

**Objective:-**

To understand and implement Least Cost Branch and Bound algorithm for solving Travelling salesperson problem and study Branch and Bound strategy.

**Theory:-**

**Travelling salesperson problem:**

Let G = *(V, E)* be a directed graph defining an instance of the traveling salesperson problem. Let *Cij* be the cost of edge <i, j>, *Cij* = ∞ if <i, j> does not belong to *E* and let | V| = *n.* Without loss of generality, we may assume that every tour starts and ends at vertex 1. So, the solution space *S* is given by *S* = {l, π, l | π is a permutation of (2, 3, ... , n)}. |*S*| = (n-1)! The size of *S* may be reduced by restricting *S* so that (1, i1, i2,... , in-1, 1) Є *S* iff <ij , ij+1 > Є E, 0 ≤ j ≤ n - 1, i0 = in= 1. *S* may be organized into a state space tree. Following figure shows the tree organization for the case of a complete graph with |V| = 4. Each leaf node *L* is a solution node and represents the tour defined by the path from the root to *L.* Node 14 represents the tour i0 = l, i1 = 3, i2 = 4, i3 = 2 and i4 = 1.

****

**Branch and Bound strategy**:

Branch and bound (BB or B&B) is an algorithm design paradigm for discrete and combinatorial optimization problems, as well as general real valued problems. A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of state space search: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root. The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.

In branch-and-bound terminology, a BFS-like state space search will be called FIFO (First In First Out) search as the list of live nodes is a first-in-first-out list (or queue). A D-searchlike state space search will be called LIFO (Last In First Out) search as the list of live nodes is a last-in-first-out list (or stack). As in the case of backtracking, bounding functions are used to help avoid the generation of subtrees that do not contain an answer node.

**Least Cost Branch and Bound:**

In order to use LC-branch-and-bound to search the traveling salesperson state space tree, we need to define a cost function c(·) and two other functions ĉ(·) and u(·) such that ĉ(R) ≤ c(R) ≤ u(R) for all nodes R. c(·) is such that the solution node with least c( ·) corresponds to a shortest tour in G. A better ĉ(·) may be obtained by using the reduced cost matrix corresponding to G.

With every node in the traveling salesperson state space tree we may associate a reduced cost matrix. Let Abe the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R,S) corresponds to including edge <i, j> in the tour. If Sis not a leaf then the reduced cost matrix for *S* may be obtained as follows:

1. Change all entries in row i and column j of A to ∞. This prevents the use of any more edges leaving vertex i or entering vertex j.

(ii) Set A(j, 1) to ∞. This prevents the use of edge <j, 1>.

(iii)Reduce all rows and columns in the resulting matrix except for rows and columns containing only ∞.

Let the resulting matrix be *B.* Steps (i) and (ii) are valid as no tour in the subtree *S* can contain edges of the type <i, k> or <k, j> or <j, 1> (except for edge <i, j>).

If *r* is the total amount subtracted in step (iii) then

ĉ (S) = ĉ (R) + A(i, j) + r.

For leaf nodes ĉ (·) =c( ) is easily computed as each leaf defines a unique tour.

For the upper bound function u, we may use u(R) = ∞ for all nodes R.

**Algorithm for TSP using LCBB Method:**

**Data Structures used:**

struct term {

int cost;

int path[10]; // 0th element of path stores no of cities traversed till that

//node and 1 onwards actual path

int matrix[10][10];

};

typedef struct term hterm; // list is an array which maintains min-heap,

// stores cost at each node, path till that

// node and reduced matrix at that node

hterm node, list[25]; // for simplicity heap is maintained from location

//one and not zero.

int visited[10]; // this array will be refreshed for each node; while

//getting expanded, stores which cities are visited till

//that node

int red-mat[10][10], temp[10][10]; // declaration for reduced matrix and

//for temporary storage

1. Read the no of cities n and read the tsp-cost matrix
2. Initialize red-matrix to tsp-cost matrix
3. cost = reduce-matrix(tsp-cost-matrix) // on this matrix perform row and column reduction and find the cost of reduction at root node.
4. node.cost  cost obtained in step 3

node.path[0] = 1; // no of cities traversed equal to one

node.path[1] =1; // started from city 1

node.matrix = reduced matrix obtained in step 3

1. node = expand(node); // perform expansion of this live node and

//get first solution

1. if node.cost < list[1].cost , go to step 14 // you are done
2. else
3. while(1) do
4. if size of heap is 0 break;
5. node = delete(); // delete the node from heap
6. node = expand(node); // again start the expansion with this

//node

1. if( node.cost < list[1].cost) go to step 14
2. end do
3. print the path using node.path
4. print cost at that node
5. you can verify the cost using original cost matrix as well
6. stop

function expansion(node)

// input is root node’s information, returns last node with path

//storing all cities and cost

1. while (1)
2. do

// store the information of this node so that it is not lost

1. count = node.path[0]; // count for no of cities traversed till

//this node

1. k= count +1; // index to store the next city to be traversed
2. cost = node.cost;
3. store node.matrix to some temporary matrix say temp-matrix // to be used for expansion
4. r = node.path[count];// last city in this path
5. for i = 1 to n set visited[i] = 0;
6. for i = 1 to count set visited[path[i]] = 1;
7. for j = 2 to n
8. Begin for
9. If( !visited[j]) // check during the expansion of

//current live node whether jth city is

//not visited

1. Begin if
2. copy the temp-matrix to red-matrix // very imp

//step

1. set-infinity(red-matrix ,r, j ) // sets all elements

//in rth row, jth column to infinity

//and element at position [j,1]

1. cost1 = reduce-matrix(red-matrix); // function

// call for row & column reduction

// initialize this node, i.e. city which was currently visited

1. node.cost = cost + cost1 + temp-matrix[r][j];
2. node.path[0] = k; // one more city visited
3. node.path[k] = j; // store the city which was

//visited

1. node.matrix = reduced matrix obtained in step16;

// store reduced matrix

1. Insert(node) // insert into heap
2. End if
3. End for
4. If( k==n) break; // all cities visited, first feasible

//solution found

1. node = delete() ; // delete from heap, this node has

//minimum cost and now it becomes

// the current node for expansion

1. End while
2. Return node

Insert and delete functions to take care of min-heap, adjusting heap, modifying size of heap.

0th location of each row and column of matrix can be used to store whether that row and column has all infinity.

**Analysis:**

While the worst case complexity of the TSP algorithms will not be any better than O(n22n), the use of good bounding functions will enable these branch-and-bound algorithms to solve some problem instances in much less time than required by the dynamic programming algorithm.

**Example:**



**State space tree generated by procedure LCBB**:



**Reduced cost matrices corresponding to nodes in state space tree shown above:**

**Input and Output:**

**Input= Cost matrix TSP graph**

**Output= Reduced matrix showing obtained by applying LCBB**

**Conclusion:**

The Least Cost Branch and Bound strategy for TSP is studied and implemented.

**FAQ :**

1. Explain Branch and Bound strategy.
2. Explain difference between backtracking and Branch and Bound.
3. Explain TSP.
4. Explain LFOBB, FIFOBB, LCBB.