Assignment - 3

Ashutosh
B21AI007

Assumptions for input:

1) Assuming that the input given is a proper formula (may be with or without brackets)
2) Each term of the formula is only a single character of uppercase or lowercase alphabets.
3) Any other format of input is not valid and may produce compilation or runtime errors while executing the code.
4) ! - Not , & - And , | - Or , > - Implication , = - Biconditional

Conversion to CNF :

1) First removing if any redundant brackets are there in the formula.
   e.g. ((p&q)) will be converted to (p&q)
2) Adding brackets to the formula, if not present in it. The brackets are add according the the precedence of operators . i.e. **! , & , | , > , =**
   e.g. !p&q will be converted to ((!p)&q)
3) Removing biconditionals in the formula by reducing it to :
   p=q → ((p>q)&(q>p))
4) Removing implications in the formula by replacing them with the given formula
   p>q = (!(p)|q)
5) Moving negation inwards by using de morgan's law and simplifying negations.
   !(a&b) – ((!a)|(!b))
   !(a|b) – ((!a)&(!b))
   !(!a) – a
6) Distributing "|" operators over "&" operators.
   e.g. ((a&b)|c) – ((a|c)&(b|c))
   ((a|(b&c)) – ((a|b)&(a|c))
7) Now we will have our formula in cnf form . Then next steps simplifies and stores it as a list of clauses as given below:
   e.g. formula (executed upto step 6) – ((a|(b|(c|d)))&(a|d))
   formula (after executing step 7) – [a|b|c|d , a|d]
   (The list of clauses shows each term of the cnf if they all are joined using an "&" operator then the give the final cnf) [:: This is done to add the terms to knowledge base while solving resolution refutation]

For example for input → **p|(q&(r>t))**
Output of CNF Conversion → **((p|q)&(p|((!r)|t)))**

Code Explanation of CNF Class :

Private Methods:

search_for_next_bracket(int i): Searches for the closing bracket corresponding to the opening bracket at index i.

search_for_prev_bracket(int i): Searches for the opening bracket corresponding to the closing bracket at index i.

add_bracket_not_operator(), add_bracket_and_operator(), add_bracket_or_operator(), add_bracket_implication(), add_bracket_biconditional(): Add brackets around subformulas based on different logical operators.

remove_implication(): Replaces implication (>) with its equivalent form in terms of other logical operators.

remove_biconditional(): Replaces biconditional (=) with its equivalent form using other logical operators.

apply_negation(int i): Applies De Morgan's laws to move negation inward.

move_negation_inward(): Moves negations inward using apply_negation.

remove_redundant_brackets(): Removes redundant parentheses.

distibute_or_over_and(): Transforms clauses to handle conjunctions.

Public Methods:

get_each_clause(int start, int end, vector<string> &v): Recursively breaks down the formula into individual clauses.

get_clause(vector<string> &v): Initializes the process of obtaining individual clauses.

get_term_of_clause(string &clause): Extracts terms within a clause, handling negation and disjunction.

convert_each_term(vector<string> &clauses): Converts each term in the obtained clauses.

convert_to_cnf(): The main method that orchestrates the entire conversion process, from removing redundant brackets to obtaining the final CNF.

Public Attributes:

vector<string> clauses: Stores the individual clauses of the CNF.

Constructor:

CNF(string &formula): Takes a logical formula as input and initializes the formula attribute.

Uninformed Search (Breadth First Search)

Used BFS as the uninformed search method to search if the query entails the knowledge base.

Breadth-First Search (BFS):

1. Start State:

The start state in BFS is the initial knowledge base represented by the vector kb before any resolution refutation steps are applied.

The knowledge base contains the clauses obtained by converting the input knowledge into Conjunctive Normal Form (CNF) and adding the negation of the query.

2. Action:

The action involves resolving pairs of clauses in the knowledge base, which starts from the initial state.

3. Successor State:

Successor states are generated by applying resolution steps to pairs of clauses in the knowledge base, leading to new configurations of the knowledge base.

4. Search Process:

The search process begins with the initial knowledge base, exploring states level by level, and attempting to resolve clauses iteratively.

5. Termination Condition (Reaching Goal State)

Termination occurs when the algorithm finds an empty resolvent clause, indicating a contradiction, or when it reaches a state where the query is entailed by the knowledge base.

Informed Search

Greedy Search:

1. Start State:

Similar to BFS, the start state in Greedy Search is the initial knowledge base, representing the vector kb before any resolution refutation steps are applied.

2. Action:

Actions involve resolving pairs of clauses in the knowledge base, starting from the initial state.

3. Successor State:

Successor states are generated by applying resolution steps to pairs of clauses in the knowledge base, incorporating a heuristic to prioritize exploration.

4. Search Process:

Greedy Search uses a priority queue to explore nodes with lower heuristic values first, emphasizing nodes that appear promising in reaching the goal quickly.

5. Termination Condition (Reaching Goal State)

Termination conditions are the same as in BFS, either finding an empty resolvent clause or reaching a state where the query is entailed by the knowledge base.

Heuristic for Greedy Search:

1. Heuristic Function:

The heuristic function used in Greedy Search aims to estimate the cost or difficulty of reaching the goal state from a given state. In this case, the heuristic is designed to prioritize states that seem closer to the goal.

2. Heuristic Value Calculation:

The heuristic value is calculated based on the last clause in the knowledge base. Specifically, it estimates the number of disjunctions ('|') present in the last clause.

The rationale behind this heuristic is that the more disjunctions present, the closer the knowledge base might be to a contradiction or resolution, potentially leading to the goal state.

3. Heuristic Application:

Nodes (states) with lower heuristic values are prioritized in the priority queue. The idea is to explore paths that seem more likely to lead to the goal state quickly.

Code Explanation :

Class Node Explanation:

The Node class is a fundamental component of the resolution refutation process, representing states in the search space. Each instance of the class corresponds to a node in the search tree. Here's a breakdown of its key components:

Attributes:

vector<string> kb: Represents the knowledge base associated with the node.
vector<Node *> children: Holds pointers to child nodes (successor states) of the current node.
bool goal: Indicates whether the node is a goal state.
string operation: Describes the operation associated with the node, providing context for the resolution refutation steps.

Constructor:

Node(vector<string> kb): Initializes a node with a given knowledge base.

Methods:

void insert(Node *next): Adds a child node (successor state) to the current node.
void set_goal(): Marks the node as a goal state.
bool is_goal_state(): Checks if the node is a goal state.
vector<string> get_kb(): Returns the knowledge base associated with the node.
vector<Node *> get_children(): Returns the children (successor states) of the node.

Breadth-First Search (BFS) Code Explanation:
The BFS code aims to explore the search space systematically, level by level, until a goal state is found or the entire space is explored. Key points to note:

Initialization:
Node *root_uninformed = new Node(kb): Creates the root node with the initial knowledge base.
queue<Node *> q: Utilizes a queue for BFS traversal.

BFS Exploration:
Nodes are explored level by level, and for each node, pairs of clauses in the knowledge base are considered for resolution.
New nodes (successor states) are created based on resolution steps and added to the queue for further exploration.
The process continues until a goal state is found or the search space is exhausted.

Result Reporting:
The number of nodes explored is tracked.
If a goal state is reached, the steps leading to the goal are printed.

Greedy Search Code Explanation:
The Greedy Search code incorporates a heuristic to prioritize exploration based on the estimated proximity to the goal state. Key points:

Initialization:
Node *root_informed = new Node(kb): Creates the root node with the initial knowledge base.
priority_queue<pair<int, Node *>, vector<pair<int, Node *>>, greater<pair<int, Node *>>> pq: Utilizes a priority queue with a heuristic for prioritized exploration.

Heuristic-Driven Exploration:
The heuristic function estimates the cost of reaching the goal state based on the number of disjunctions in the last clause of the knowledge base.
Nodes are prioritized in the priority queue based on their heuristic values.
Lower heuristic values indicate nodes that are likely closer to the goal.

Result Reporting:
The number of nodes explored is tracked.

If a goal state is reached, the steps leading to the goal are printed.

Running On Different Inputs

Input 1 :

```
4 1
p|(q&(r>t))
p>r
q>t
q>(r=t)
r
```

Output :

```
Knowledge base after adding negation of query to it and converting to CNF
:
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)

Executing uninformed search (Breadth First Search) ....
Query entails the knowledge base.
Number of Nodes Explored : 187
Number of Steps required : 4

p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)



Resolving p|q and (!p)|r and addding q|r to knowledge base
p|q
p|(!r)|t
```

```
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r


Resolving (!q)|t and (!q)|(!t)|r and addding (!q)|r to knowledge base
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r
(!q)|r


Resolving q|r and (!q)|r and addding r to knowledge base
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r
(!q)|r
r


Have contradiction in knowledge base if we resolve (!r) and r
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
```

```
(!r)
q|r
(!q)|r
r



Time taken by uninformed search is 37 milliseconds


Executing Greedy Search ....
Query entails the knowledge base.
Number of Nodes Explored : 29
Number of Steps required : 4

p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)


Resolving p|q and (!p)|r and addding q|r to knowledge base
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r


Resolving (!q)|t and (!q)|(!t)|r and addding (!q)|r to knowledge base
p|q
p|(!r)|t
(!p)|r
(!q)|t
```

```
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r
(!q)|r


Resolving q|r and (!q)|r and addding r to knowledge base
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r
(!q)|r
r


Have contradiction in knowledge base if we resolve (!r) and r
p|q
p|(!r)|t
(!p)|r
(!q)|t
(!q)|(!r)|t
(!q)|(!t)|r
(!r)
q|r
(!q)|r
r



Time taken by greedy search is 15 milliseconds
```

Input 2 :

```
7 0
a>b
b>c
c>d
d>e
e>f
f>g
g>h
a>h
```

Output :

```
Knowledge base after adding negation of query to it and converting to CNF
:
(!p)|q
(!q)|r
(!r)|s
(!s)|t
(!t)|u
(!u)|v
(!v)|w
p
(!w)

Executing uninformed search (Breadth First Search) ....
Query entails the knowledge base.
Number of Nodes Explored : 28962
Number of Steps required : 8
Time taken by uninformed search is 1933 milliseconds


Executing Greedy Search ....
Query entails the knowledge base.
Number of Nodes Explored : 24
Number of Steps required : 8
```

```
Time taken by greedy search is 4 milliseconds
```

Input 3:

```
2 1
A=(B|C)
!A
!B
```

Output:

```
Knowledge base after adding negation of query to it and converting to CNF
:
(!A)|B|C
(!B)|A
(!C)|A
(!A)
B

Executing uninformed search (Breadth First Search) ....
Query entails the knowledge base.
Number of Nodes Explored : 6
Number of Steps required : 2

(!A)|B|C
(!B)|A
(!C)|A
(!A)
B


Resolving (!B)|A and (!A) and addding (!B) to knowledge base
(!A)|B|C
(!B)|A
(!C)|A
(!A)
B
(!B)


Have contradiction in knowledge base if we resolve B and (!B)
(!A)|B|C
(!B)|A
```

```
(!C)|A
(!A)
B
(!B)



Time taken by uninformed search is 6 milliseconds


Executing Greedy Search ....
Query entails the knowledge base.
Number of Nodes Explored : 3
Number of Steps required : 2

(!A)|B|C
(!B)|A
(!C)|A
(!A)
B


Resolving (!B)|A and B and addding A to knowledge base
(!A)|B|C
(!B)|A
(!C)|A
(!A)
B
A


Have contradiction in knowledge base if we resolve (!A) and A
(!A)|B|C
(!B)|A
(!C)|A
(!A)
B
A
```

```
Time taken by greedy search is 5 milliseconds
```

Input 4 :

```
3 1
(p>q)>q
(p>p)>r
(r>s)>!(s>q)
r
```

Output:

```
Knowledge base after adding negation of query to it and converting to CNF
:
p|q
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
(!s)|(!q)
(!r)

Executing uninformed search (Breadth First Search) ....
Query entails the knowledge base.
Number of Nodes Explored : 8
Number of Steps required : 2

p|q
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
(!s)|(!q)
(!r)



Resolving p|r and (!p)|r and addding r to knowledge base
p|q
```

```
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
(!s)|(!q)
(!r)
r


Have contradiction in knowledge base if we resolve (!r) and r
p|q
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
(!s)|(!q)
(!r)
r



Time taken by uninformed search is 10 milliseconds


Executing Greedy Search ....
Query entails the knowledge base.
Number of Nodes Explored : 6
Number of Steps required : 2

p|q
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
```

```
(!s)|(!q)
(!r)



Resolving p|r and (!p)|r and addding r to knowledge base
p|q
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
(!s)|(!q)
(!r)
r



Have contradiction in knowledge base if we resolve (!r) and r
p|q
(!q)|q
p|r
(!p)|r
r|s
(!q)|r
(!s)|s
(!s)|(!q)
(!r)
r




Time taken by greedy search is 9 milliseconds
```

Input 5 :

```
4 1
p|(q&(r>t))
p>q
q>t
```

```
!r&(p&q)
r
```

Output:

```
Knowledge base after adding negation of query to it and converting to CNF
:
p|q
p|(!r)|t
(!p)|q
(!q)|t
(!r)
&
(
q
(!r)

Executing uninformed search (Breadth First Search) ....
Query does not entails the knowledge base.
Time taken by uninformed search is 2 milliseconds


Executing Greedy Search ....
Query does not entails the knowledge base.
Time taken by greedy search is 2 milliseconds
```

Input	Number of Nodes Explored with bfs	Number of Steps required with bfs	Time taken wit bfs (uninformed search)	Number of Nodes Explored with greedy search	Number of steps required with greedy	Time taken with greedy search
4 1 p|(q&(r>t)) p>r q>t q>(r=t) r	187	4	37 ms	29	4	15 ms
7 0 a>b b>c c>d d>e e>f f>g g>h a>h	28962	8	1933 ms	24	8	4 ms
2 1 A=(B|C) !A !B	6	2	6 ms	3	2	5 ms
3 1 (p>q)>q (p>p)>r (r>s)>!(s>q) r	8	2	10 ms	6	2	9 ms
4 1 p|(q&(r>t)) p>q q>t	Query	Does	Not	Entails	Knowledge	Base

!r&(p&q) r						

Observations:

Efficiency Comparison:

In most test cases, the execution time of Greedy Search is less than or equal to that of Breadth-First Search (BFS). This suggests that the Greedy Search strategy tends to be more time-efficient in resolving the entailment or contradiction with the query.

Similarly, the number of nodes explored in Greedy Search is generally less than or equal to BFS. This indicates that Greedy Search explores a more focused set of nodes, making it more efficient in terms of space utilization.

Consistency in Steps:

The number of steps required to reach the answer is consistent between Greedy Search and BFS for most test cases. This suggests that the resolution steps involved in both strategies are similar in terms of logical inference, regardless of the search strategy used.

Non-linear Relationship with 'n':

The relationship between the number of nodes explored and the size of the knowledge base (n) is not strictly linear. The number of nodes explored is influenced by the actual input content and the reduction of each formula size after resolution which is clear from the given example inputs in the table.

Dependence on Input Characteristics:

The observations highlight the importance of understanding the actual content and characteristics of the input. The efficiency of the search strategies is not solely determined by the size of the knowledge base but also by its specific features.

Analysis:

Greedy vs. BFS Comparison:

In most cases, the time taken for Greedy Search is less than or equal to that of Breadth-First Search (BFS).

Similarly, the number of nodes explored in Greedy Search is generally less than or equal to BFS.

The number of steps required to reach the answer is comparable between Greedy Search and BFS.

Knowledge Base Entailment:

When the knowledge base entails the query, both Greedy Search and BFS successfully find the answer.

The number of nodes explored and the time taken are reasonable for both algorithms in such cases.

Knowledge Base Non-Entailment:
In situations where the knowledge base does not entail the query, both algorithms need to explore the entire search space.
The number of nodes explored is similar for both Greedy Search and BFS when the knowledge base cannot entail the query.

Impact of Input Size:
The relationship between the input size (number of formulas in the knowledge base) and the number of nodes explored is not strictly linear.
For example, in some cases, increasing the input size (n) may not necessarily result in a proportional increase in the number of nodes explored.

Efficiency of Greedy Search:
Greedy Search tends to be more efficient, especially in scenarios with larger input sizes (e.g., n = 6 and n = 7), where the time taken and number of nodes explored are significantly less compared to BFS.
The heuristic used in Greedy Search, which prioritizes nodes with fewer terms in the knowledge base, appears effective in reaching the goal state efficiently.

Contradictions and Resolution:
The algorithms successfully identify contradictions in the knowledge base and resolve them, leading to the conclusion that the query is entailed or not.

Execution Time Observations:
Execution times for both Greedy Search and BFS are relatively low, even for larger input sizes, suggesting the efficiency of the implemented algorithms.

Algorithm Completeness:
Both Greedy Search and BFS demonstrate completeness, providing reasonable results for both entailment and non-entailment scenarios.