Iterative deepening dfs:

Use Example: Imagine a puzzle where you need to move a robot through a maze to find a goal. You want to find the shortest path for the robot to reach the goal. You can use Iterative Deepening Search to systematically explore paths of increasing depths until the goal is found.

```
function iterative_deepening_search(problem):
   for depth in range(0, infinity):
      result = depth_limited_search(problem, depth)
      if result == "goal found" or result == "cutoff":
         return result

function depth_limited_search(problem, depth):
   return recursive_dls(problem.initial_state, problem, depth)

function recursive_dls(node, problem, depth):
   if problem.is_goal(node.state):
      return "goal found"
   elif depth == 0:
      return "cutoff"
   else:
      cutoff_occurred = false
      for child in node.expand(problem):
         result = recursive_dls(child, problem, depth - 1)
         if result == "cutoff":
            cutoff_occurred = true
         elif result != "goal not found":
            return result
      return "cutoff" if cutoff_occurred else "goal not found"
```

A* search algorithm:

Use Example: Consider a map where you want to find the shortest path from one city to another. You can use A* search to efficiently explore routes based on both the actual distance traveled and a heuristic estimate of the remaining distance to the destination.

```
function A_star_search(problem):
   initialize start node
   create a priority queue and add the start node with priority f(start)
   while priority queue is not empty:
      node = remove node from priority queue
      if node is goal:
         return reconstruct_path(node)
      for each neighbor in node.neighbors:
         if neighbor not in explored or neighbor in priority queue with lower f-value:
            update priority queue with neighbor
   return "goal not found"
```

```
function f(node):
   return g(node) + h(node)

function g(node):
   return cost from start to node

function h(node):
   return heuristic from node to goal

function reconstruct_path(node):
   path = []
   while node is not null:
      path.insert(0, node)
      node = node.parent
   return path
```

3. Vacuum Cleaner Agent:

Use Example: In a smart home scenario, you have a vacuum cleaner robot that needs to clean all the rooms. The Vacuum Cleaner Agent pseudocode illustrates how the robot can efficiently move through rooms, cleaning them as needed.

```
function vacuum_cleaner_agent():
   while any_dirty_room_exists():
      current_room = sense_current_room()
      if is_dirty(current_room):
         clean_current_room()
      else:
         move_to_next_room()

function any_dirty_room_exists():
   return any(room.is_dirty for room in environment.rooms)

function sense_current_room():
   return environment.current_room

function is_dirty(room):
   return room.is_dirty

function clean_current_room():
   environment.clean_room()

function move_to_next_room():
   environment.move_to_next_room()
```

4 Knowledge Base and Entailment Check (Propositional Logic):

Use Example: Imagine a knowledge base that contains information about the state of a traffic light and a query asking whether it entails that it is safe to cross the road. The Knowledge Base and Entailment Check pseudocode can be used to verify if the information supports the query.

```
# Define your knowledge base (KB) using propositional logic statements
knowledge_base = [...]

# Define the query
query = [...]

# Check if the query entails the knowledge base
result = check_entailment(knowledge_base, query)
print("The query entails the knowledge base." if result else "The query does not entail the knowledge base.")

# Entailment check function
function check_entailment(knowledge_base, query):
    # Your entailment checking logic here
    # ...

    # Return True if the query entails the knowledge base, otherwise False
    return ...
```

5 Knowledge Base and Resolution (Propositional Logic):

Use Example: Suppose you have a knowledge base about relationships between people, and you want to prove that someone's uncle is also their father's brother. You can use the Knowledge Base and Resolution pseudocode to show that this relationship is logically entailed.

```
# Define your knowledge base (KB) using propositional logic statements
knowledge_base = [...]

# Define the query
query = [...]

# Prove the query using resolution
result = resolution_prove(knowledge_base, query)
print("The query is proven." if result else "The query is not proven.")
```

```
# Resolution proving function
function resolution_prove(knowledge_base, query):
    # Your resolution proving logic here
    # ...

    # Return True if the query is proven, otherwise False
    return ...
```

6 Unification in First Order Logic:

Use Example: Consider a scenario where you want to unify statements like "P(x, y)" and "P(a, b)" in first-order logic. The Unification pseudocode can be applied to find a substitution (theta) that makes these statements equivalent.

```
function unification(theta, alpha, beta):
    # Your unification logic here
    # ...

# Example usage
theta = {}
alpha = parse("P(x, y)")
beta = parse("P(a, b)")
result = unification(theta, alpha, beta)
print("Unification successful." if result else "Unification failed.")
```

7 Convert First Order Logic Statement to CNF:

Use Example: Suppose you have a first-order logic statement involving implications and universal quantifiers. You can use the Convert to CNF pseudocode to transform this statement into Conjunctive Normal Form (CNF) for easier handling.

```
function convert_to_cnf(statement):
    # Your CNF conversion logic here
    # ...

# Example usage
statement = parse("forall x (P(x) => Q(x))")
cnf_result = convert_to_cnf(statement)
print("Converted CNF:", cnf_result)
```

8 Forward Reasoning (First Order Logic):

Use Example: Consider a knowledge base with statements about mortality and a query asking whether a specific individual, like Socrates, is mortal. The Forward Reasoning pseudocode can be applied to prove or disprove the query based on the provided knowledge.

```
function forward_reasoning(knowledge_base, query):
   # Your forward reasoning logic here
   # ...

# Example usage
knowledge_base = [...]
query = "mortal(Socrates)"
result = forward_reasoning(knowledge_base, query)
print("The query is proven." if result else "The query is not proven.")
```

9 Greedy Best-First Search:

Use Example: Imagine a scenario where you are navigating through a maze, and you want to use a heuristic that guides you based on the distance to the goal only. Greedy Best-First Search can be applied to efficiently explore paths with a focus on reaching the goal quickly.

```
function greedy_bfs(problem):
   create a priority queue and add the start node with priority h(start)
   while priority queue is not empty:
      node = remove node from priority queue
      if node is goal:
         return reconstruct_path(node)
      for each neighbor in node.neighbors:
         if neighbor not in explored or neighbor in priority queue with lower h-
value:
            update priority queue with neighbor
   return "goal not found"
```

10 Depth-First Search:

Use Example: Consider a problem where you are searching for a specific item in a large space. Depth-First Search can be used to systematically explore the space, going deep into one path before backtracking to explore other possibilities.

```
function dfs(node, problem):
   if problem.is_goal(node.state):
```

```
        return reconstruct_path(node)
    for child in node.expand(problem):
        result = dfs(child, problem)
        if result is not "goal not found":
            return result
    return "goal not found"
```

11 tic tac toe

Use Example: For a Tic-Tac-Toe game, you can use the Minimax Algorithm to determine the best move for a player. The algorithm considers all possible moves and their outcomes to find the optimal move for a given board state.

```
# Define the player symbols
PLAYER_X = 'X'
PLAYER_O = 'O'

function minimax(board, depth, maximizing_player):
    if is_terminal(board):
        return evaluate(board)

    if maximizing_player:
        max_eval = -infinity
        for each move in possible_moves(board):
            eval = minimax(resulting_board_after_move(board, move), depth + 1,
False)
            max_eval = max(max_eval, eval)
        return max_eval
    else:
        min_eval = +infinity
        for each move in possible_moves(board):
            eval = minimax(resulting_board_after_move(board, move), depth + 1,
True)
            min_eval = min(min_eval, eval)
        return min_eval

function is_terminal(board):
    # Check if the game is over (someone won or it's a draw)
    # ...

function evaluate(board):
    # Evaluate the current board state
```

```
    # ...

function possible_moves(board):
    # Generate a list of possible moves for the current board
    # ...

function resulting_board_after_move(board, move):
    # Apply the move to the current board and return the resulting board
    # ...
```