AI Sous-Chef Project Report

Team ID: 47
Team Members:

1. [Name] - [Student ID]
2. [Name] - [Student ID]
3. [Name] - [Student ID]

Date: December 5th, 2025

Code Implementation with Detailed Comments

1. Knowledge Base Inclusion

```
:- include('KB1.pl').
```

Comment: Loads the ordering constraints (`above/2` facts) from the knowledge base. Can be switched to KB2.pl for testing different constraint sets.

---

2. SUCCESSOR STATE AXIOM: `stacked/2`

This is the **only fluent** and **only successor state axiom** in our implementation.

```
stacked(Ingredient, result(Action, S)) :-
    Action = stack(Ingredient)
    ;
    stacked(Ingredient, S).

stacked(_, s0) :- fail.
```

PROF

ARGUMENTS:

- **`Ingredient`**: An atom representing the ingredient
    - **Type**: Atom
    - **Domain**: `{bottom_bun, patty, lettuce, cheese, pickles, tomato, top_bun}`
    - **Purpose**: Identifies which ingredient we're checking for in the situation
- **`S`** (or `result(Action, S)`): A situation term representing the world state
    - **Type**: Situation term
    - **Domain**: `s0` (initial state) or `result(Action, PreviousSituation)`

- **Purpose**: Represents the state of the world at a particular point in the action sequence

SUCCESSOR STATE AXIOM FORM:

This follows the standard SSA pattern: $\texttt{Fluent(result(a,s))} \leftrightarrow \gamma^{+}\texttt{(a,s)} \lor \texttt{Fluent(s)}$

1. **Positive Effect Axiom (γ^+)**: `Action = stack(Ingredient)`
    - The ingredient becomes stacked if the action performed is `stack(Ingredient)`
    - This is how actions change the world state
2. **Frame Axiom (Persistence)**: `stacked(Ingredient, S)`
    - The ingredient remains stacked if it was already stacked in the previous situation
    - This solves the frame problem - things persist unless explicitly changed
3. **Base Case**: `stacked(_, s0) :- fail`
    - No ingredients are stacked in the initial situation s0
    - Establishes the starting state of the world

Why this is a proper SSA:

- Explicitly checks the `Action` parameter (not just recursion)
- Includes both effect axiom and frame axiom
- Has proper base case
- Uses pure situation calculus (no assert/retract)

---

3. Helper Predicate: `all_ingredients_stacked/1`

```
all_ingredients_stacked(S) :-
    stacked(bottom_bun, S),
    stacked(patty, S),
    stacked(lettuce, S),
    stacked(cheese, S),
    stacked(pickles, S),
    stacked(tomato, S),
    stacked(top_bun, S).
```


ARGUMENT:

- `S`: The situation to check

Comment: Verifies all 7 required ingredients are present by checking the `stacked/2` fluent for each. All checks must succeed (conjunction). Ensures burger completeness.

---

4. Helper Predicate: `count_stacks/3`

```
count_stacks(_, s0, 0).

count_stacks(Ingredient, result(stack(Ingredient), S), Count) :-
    count_stacks(Ingredient, S, PrevCount),
    Count is PrevCount + 1.

count_stacks(Ingredient, result(stack(Other), S), Count) :-
    Other \= Ingredient,
    count_stacks(Ingredient, S, Count).
```

ARGUMENTS:

- `Ingredient`: The ingredient to count
- `S`: The situation to examine
- `Count`: Output - number of times the ingredient was stacked

Comment: Recursively traverses the situation structure backwards from the final state to s0. When it finds the ingredient being stacked, it increments the count. When it finds a different ingredient, it maintains the count. Essential for detecting duplicates (Query 5).

---

5. Helper Predicate: `one_of_each_ingredient/1`

```
one_of_each_ingredient(S) :-
    count_stacks(bottom_bun, S, 1),
    count_stacks(patty, S, 1),
    count_stacks(lettuce, S, 1),
    count_stacks(cheese, S, 1),
    count_stacks(pickles, S, 1),
    count_stacks(tomato, S, 1),
    count_stacks(top_bun, S, 1).
```

PROF

ARGUMENT:

- `S`: The situation to validate

Comment: Ensures each ingredient appears exactly once by verifying `count_stacks/3` returns 1 for each ingredient. Prevents duplicate ingredients. Critical for rejecting Query 5 (duplicate cheese).

---

6. Helper Predicate: `first_action/2`

```
first_action(result(Action, s0), Action).

first_action(result(_, S), Action) :-
    first_action(S, Action).
```

ARGUMENTS:

- S: The situation
- Action: Output - the first action performed from s0

Comment: Recursively searches backwards through the situation until it reaches s0, then returns the action immediately after s0. Used to verify bottom_bun is the first ingredient stacked.

---

7. Helper Predicate: last_action/2

```
last_action(result(Action, _), Action).
```

ARGUMENTS:

- S: The situation (must be in form result(Action, _))
- Action: Output - the last (most recent) action

Comment: Simply returns the outermost action from the situation structure. No recursion needed since the outermost action is the most recent. Used to verify top_bun is the last ingredient stacked. Critical for rejecting Query 4.

---

8. Helper Predicate: stacked_before/3

```
stacked_before(Ingredient1, Ingredient2, result(stack(Ingredient2), S))
:-
    stacked(Ingredient1, S),
    !.

stacked_before(Ingredient1, Ingredient2, result(stack(_), S)) :-
    stacked_before(Ingredient1, Ingredient2, S).
```


ARGUMENTS:

- Ingredient1: The ingredient that should be below (stacked earlier)
- Ingredient2: The ingredient that should be above (stacked later)
- S: The situation to examine

Comment: Traverses the situation backwards. When it finds Ingredient2 being stacked, it checks if Ingredient1 was already stacked in the previous situation. The cut (!) stops the search once found for efficiency. This implements temporal ordering - if Ingredient1 was stacked before Ingredient2, then Ingredient1 is physically below Ingredient2 in the burger.

---

9. Helper Predicate: constraints_satisfied/1

```
constraints_satisfied(S) :-
    forall(
        above(Upper, Lower),
        (stacked(Upper, S), stacked(Lower, S), stacked_before(Lower,
Upper, S))
    ).
```

ARGUMENT:

- `S`: The situation to validate

Comment: Validates all ordering constraints from the knowledge base. Uses `forall/2` to check every `above(Upper, Lower)` fact. For each constraint: (1) both ingredients must be stacked, and (2) `Lower` must have been stacked before `Upper` (i.e., `Lower` is below `Upper` in the burger). If any constraint fails, the entire predicate fails.

---

10. Goal Predicate: `burgerReady_goal/1`

```
burgerReady_goal(S) :-
    all_ingredients_stacked(S),
    one_of_each_ingredient(S),
    first_action(S, stack(bottom_bun)),
    last_action(S, stack(top_bun)),
    constraints_satisfied(S).
```

ARGUMENT:

- `S`: A situation to validate as a complete burger

Comment: Defines the goal state for IDS. All five criteria must be satisfied:

PROF

1. All 7 ingredients present
2. Each ingredient appears exactly once (no duplicates)
3. Bottom bun is the first action
4. Top bun is the last action
5. All `above/2` constraints from KB are satisfied

This is the validation function that determines if a candidate solution is valid.

---

11. Situation Generator: `build_situation/1`

```
build_situation(s0).

build_situation(result(stack(Ingredient), S)) :-
    build_situation(S),
```

```
    member(Ingredient, [bottom_bun, patty, lettuce, cheese, pickles,
tomato, top_bun]),
    \+ stacked(Ingredient, S).
```

ARGUMENT:

- `S`: A situation built from s0 through stacking actions

Comment: Generates candidate situations for IDS to explore. Recursively builds situations by choosing ingredients and stacking them. `member/2` provides backtracking to try different ingredients. The `\+ stacked(Ingredient, S)` check ensures no ingredient is stacked twice during generation. This creates the search space - all possible orderings of 7 ingredients.

---

12. MAIN PREDICATE: `burgerReady/1`

```
burgerReady(S) :-
    ids_burgerReady(S, 10).
```

ARGUMENT:

- `S`: Either unbound (to find a solution) or bound (to validate a configuration)

Comment: Main entry point for the system. Uses **ONLY** Iterative Deepening Search - no other search method is used. Calls IDS with an initial depth limit of 10, which is sufficient for stacking 7 ingredients plus validation checks.

---

13. IDS IMPLEMENTATION: `ids_burgerReady/2`

PROF

```
ids_burgerReady(S, Limit) :-
    call_with_depth_limit(
        (build_situation(S), burgerReady_goal(S)),
        Limit,
        Result
    ),
    (   number(Result) -> true
    ;   Result = depth_limit_exceeded,
        NewLimit is Limit + 1,
        ids_burgerReady(S, NewLimit)
    ).
```

ARGUMENTS:

- `S`: The situation to find or validate
- `Limit`: Current depth limit for the search

Comment: Implements Iterative Deepening Search using `call_with_depth_limit/3` as required by the assignment. The algorithm:

1. Attempts to prove `(build_situation(S), burgerReady_goal(S))` within the current depth limit
2. If `Result` is a number: solution found at that depth, return success
3. If `Result` is `depth_limit_exceeded`: increment limit by 1 and recursively try again

This ensures completeness - if a solution exists, IDS will eventually find it by systematically increasing the depth. The incremental approach (Limit + 1) ensures we find solutions at the shallowest depth first.
