Project Report: AI Sous-Chef
Logic-Based Agent using Situation Calculus

Team 10
Abdullah Sherif
Youssef Fouda
Omar Orensa

November 25, 2025

Abstract

This report documents the design and implementation of an AI agent acting as a "Sous-Chef" for burger assembly. Using Prolog and the Situation Calculus framework, we developed an agent capable of reasoning about action effects and constraints to construct valid burgers. A key focus of our implementation was the use of Successor State Axioms (SSAs) to manage state changes without destructive database modifications, and the adoption of Iterative Deepening Search (IDS) to overcome the completeness limitations of standard Depth-First Search.

1 Introduction

The goal of this assignment was to simulate a reasoning agent capable of assembling a burger from a set of ingredients. Unlike a static recipe, our agent must derive the correct order of ingredients dynamically based on a Knowledge Base (KB) of partial constraints (e.g., `above(lettuce, patty)`).

We approached this problem using **Situation Calculus**, a logic formalism that represents changing worlds. Instead of simply storing the current state of the burger, we model the "history" of actions (situations). The agent must find a sequence of `stack(X)` actions that satisfies all topological constraints defined in the KB.

2 Implementation Strategy

Our solution relies on three main architectural components: Fluents to describe the world, Successor State Axioms to describe changes, and a Search Strategy to find the goal.

2.1 Fluents and State Representation

We defined "fluents"—predicates that change over time—to track the progress of the burger assembly.

- `stacked(X, S)`: True if ingredient `X` is present in the stack in situation `S`.
- `onTop(X, S)`: True if ingredient `X` is the topmost item in situation `S`.

2.2 Successor State Axioms (SSAs)

A critical requirement of the assignment was to avoid `assert` or `retract`. We achieved this by implementing Successor State Axioms. These axioms define true statements in the *next* situation based on the *current* situation and the action performed.

For example, our SSA for `stacked/2` ensures that once an item is stacked, it remains stacked in all future situations:

```
% An ingredient X is stacked in the result of stacking A if:
% 1. X is the ingredient A being stacked right now.
% 2. OR X was already stacked in the previous situation S.
stacked(X, result(stack(A), S)) :-
    X = A ;
```

```
    (stacked(X, S), X \= A).
```

Listing 1: Successor State Axiom for stacked

This logic allows the agent to "remember" the burger's composition purely through logical deduction.

2.3 Action Preconditions

To make the search efficient, we restricted when an action is valid. The predicate `poss(stack(X), S)` defines the preconditions. We decided that an ingredient `X` can only be stacked if:

1. It hasn't been used yet (preventing duplicates).
2. **Critical Step:** All ingredients required to be *below* `X` (according to the `above` rules) are already in the stack.

This "look-ahead" constraint significantly prunes the search tree. By checking dependencies *before* stacking, we prevent the agent from building invalid partial burgers that would require backtracking later.

2.4 Search Algorithm: Iterative Deepening (IDS)

Prolog's standard reasoning mechanism is Depth-First Search (DFS). However, in planning problems, DFS often dives down infinite branches or suboptimal paths. To guarantee that we find a solution (and a short one), we implemented **Iterative Deepening Search (IDS)**.

We created a `solve/1` predicate that incrementally increases the search depth from 1 to 20. This ensures completeness: if a solution exists, our agent will find it.

```
solve(Solution) :-
    between(1, 20, Depth),        % Try depths 1, 2, ... 20
    search(s0, Depth, Solution),  % Attempt to find solution at this depth
    !.                            % Stop immediately once found
```

Listing 2: Iterative Deepening Search Implementation

3 Testing and Results

We validated our agent against two distinct Knowledge Bases provided in the assignment.

3.1 Test Case 1: KB1 Constraints

Constraints:

- `above(tomato, lettuce)`
- `above(tomato, patty)`
- `above(pickles, cheese)`
- `above(cheese, tomato)`

Outcome: The agent correctly deduced that the dependency chain is *lettuce/patty* $\rightarrow$ *tomato* $\rightarrow$ *cheese* $\rightarrow$ *pickles*. It generated a plan placing the foundation ingredients first, satisfying all constraints.

3.2 Test Case 2: KB2 Constraints

Constraints:

- `above(lettuce, patty)`
- `above(cheese, lettuce)`
- `above(pickles, tomato)`

Outcome: Even though the constraints in KB2 were "looser" (fewer inter-dependencies), our agent successfully constructed a valid burger. The precondition logic robustly handled the different ordering without requiring any code changes.

4 Conclusion

In this project, we successfully implemented a logic-based planner for burger assembly. By adhering to Situation Calculus principles, we created a system that is both declarative and flexible. The use of Successor State Axioms allowed us to model state changes rigorously, while Iterative Deepening Search provided a robust mechanism to guarantee solution discovery. The agent passes all provided test cases, demonstrating that it can dynamically adapt to different "recipes" defined in the Knowledge Base.