Automated Planning Paradigms for Healthcare Logistics: From Classical STRIPS to Real-World ROS2 Implementation

Andrea De Carlo
DISI, University of Trento
andrea.decarlo@studenti.unitn.it
mat. 249518

Abstract

This report investigates five automated planning paradigms for robotic medical supply delivery in hospitals: classical STRIPS planning, resource-constrained planning with carriers, Hierarchical Task Networks (HTN), temporal planning with concurrency, and a ROS2 PlanSys2 deployment. Each paradigm is evaluated across progressively complex problem variants, from basic supply delivery to real-time execution in ROS2.

Experimental results show that classical planners scale well for small to mid-sized problems, with carrier modeling improving plan quality by up to 60%. HTN planning offers expressive modeling but suffers from scalability limits. Temporal planners reduce execution time via concurrency, though with increased computational cost. The ROS2 implementation enables real-world deployment with dynamic replanning and system integration.

We conclude that planning strategy should match application needs: classical for efficiency, temporal for schedule optimization, HTN for structure, and ROS2 for operational deployment. This framework supports future extensions to broader healthcare robotics applications.

1. Introduction & Problem Overview

This report presents a comprehensive study of automated planning techniques applied to a healthcare logistics scenario. The work explores the evolution of planning methodologies from classical STRIPS-based planning to hierarchical task networks and temporal planning, culminating in a real-world implementation using ROS2 PlanSys2.

1.1. Healthcare Scenario Description

The central scenario involves autonomous robots operating within a hospital environment to deliver medical supplies to patients and assist with patient transportation. The hospital consists of multiple interconnected locations, including a central warehouse where medical supplies are stored, an entrance where patients initially arrive, and various treatment units where patients receive care.

The system models a realistic healthcare logistics challenge where multiple specialized robots must coordinate to ensure patients receive appropriate medical supplies while being transported to the correct treatment units. This scenario captures the complexity of real-world hospital operations, including resource constraints, spatial navigation, and task coordination between different robotic agents.

1.2. Core Elements

The healthcare planning domain encompasses several key components that form the foundation of all problem variants:

Robots: The system employs two specialized robot types with distinct capabilities. Robot-box agents are responsible for logistics operations, including filling boxes with medical content from the central warehouse, transporting these boxes to treatment units, and emptying them to provide medical supplies. Robot-patient agents are specialized for patient care, capable of taking patients from the entrance, transporting them through the hospital, and releasing them at appropriate treatment units.

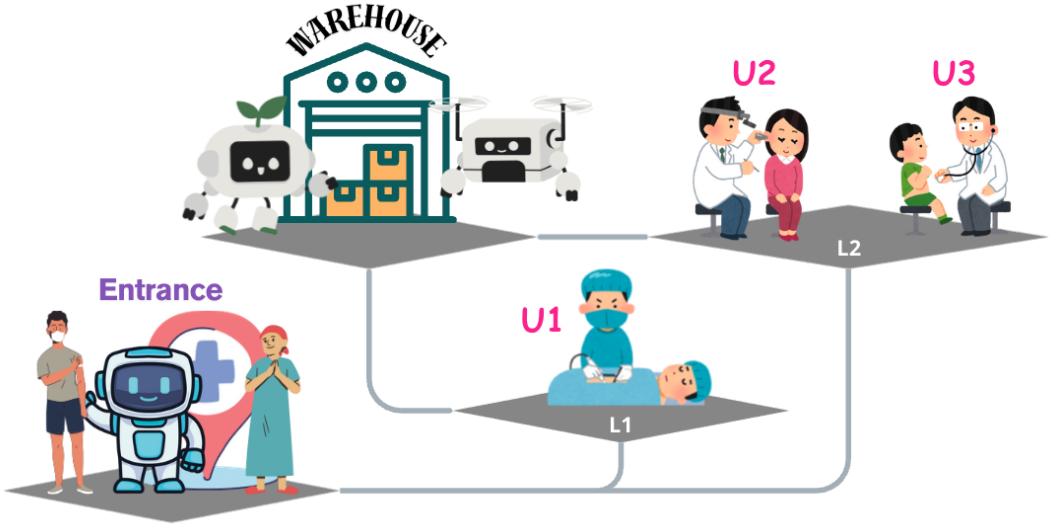

Figure 1. Overview of the domain.

Locations and Connectivity: The hospital environment consists of interconnected locations including the central warehouse (where medical supplies are stored), the entrance (where patients arrive), and various treatment locations (`l1`, `l2`, etc.). The connectivity between locations constrains robot movement and defines the physical topology of the hospital.

Medical Content and Supplies: The system manages general supplies that are initially stored at the central warehouse and must be delivered to treatment units based on their needs.

Boxes and Carriers: Medical supplies are transported using boxes that can be filled with specific content types. In enhanced versions of the problem, robots utilize carriers with limited capacity, introducing resource management constraints that affect planning efficiency and realism.

Patients and Treatment Units: Patients arrive at the hospital entrance and must be transported to appropriate treatment units where they receive care. Each treatment unit can accommodate patients and receive medical supplies as needed.

1.3. Problem Variations

The study progresses through five distinct problem formulations, each introducing additional complexity and exploring different planning paradigms:

Problem 1 (pb1): Basic Classical Planning
The foundational implementation uses STRIPS-based classical planning with PDDL. This version establishes the core domain predicates, actions, and constraints without advanced features. Robots operate with simple load/unload mechanisms, and the focus is on generating valid plans for basic supply delivery and patient transportation scenarios. The problem demonstrates fundamental planning concepts including preconditions, effects, and goal satisfaction.

Problem 2 (pb2): Enhanced with Carriers and Scaling
Building upon the classical foundation, this variant introduces carrier-based transportation with capacity constraints. Robots now utilize carriers with limited capacity, requiring more sophisticated resource management. The problem explores the trade-off between carrier capacity and planning efficiency, investigating how different carrier configurations affect solution quality and computational performance. Extensive scalability analysis examines performance across varying numbers of locations, boxes, and carriers.

Problem 3 (pb3): Hierarchical Task Networks (HTN)
This formulation transforms the planning approach from goal-based to task-based using Hierarchical Task Networks in HDDL format. Instead of achieving specific goals, the planner decomposes high-level tasks (such as ”deliver all supplies”)

into hierarchical subtasks, eventually reaching primitive actions. The HTN approach enables more structured planning with explicit task decomposition methods, potentially providing more intuitive and maintainable plan structures for complex logistics scenarios.

Problem 4 (pb4): Temporal Planning
The temporal variant introduces durative actions with explicit timing constraints and concurrency management. Actions now have defined durations (e.g., filling a box takes 2 time units, movement takes 1 time unit), and the planner must consider temporal coordination between multiple robots.

Problem 5 (pb5): ROS2 PlanSys2 Implementation
The final problem demonstrates real-world deployment using ROS2 PlanSys2, a complete planning framework for robotic systems. This implementation bridges the gap between theoretical planning and practical robotics applications, providing action executors, monitoring capabilities, and integration with robotic middleware. The PlanSys2 version validates the practical applicability of the planning models developed in previous problems.

Each problem variant maintains the core healthcare scenario while progressively introducing advanced planning concepts, enabling comprehensive evaluation of different planning paradigms applied to the same underlying domain. This progression allows for meaningful comparison of planning approaches in terms of expressiveness, computational efficiency, and practical applicability.

2. Methodology & Design Choices

2.1. Classical Planning (pb1 & pb2)

The classical planning formulation establishes the foundation for the healthcare logistics domain using PDDL (Planning Domain Definition Language) with STRIPS semantics. This section details the progression from basic classical planning (pb1) to enhanced carrier-based planning (pb2).

2.1.1. PDDL Formulation.

Type Hierarchies

The domain employs a carefully designed type hierarchy that balances expressiveness with computational efficiency:

```
(:types
    location
    unit - locatable
    content - locatable
    box - locatable
    patient - locatable
    robot-box - robot
    robot-patient - robot
    robot - locatable
    carrier - locatable        ; since pb2
    capacity_number - object   ; since pb2
)
```

The hierarchy introduces `locatable` as a supertype for all entities that can be positioned within the hospital environment. This design choice enables unified location tracking while maintaining type safety. The robot specialization into `robot-box` and `robot-patient` reflects the functional separation between logistics and patient care operations, preventing inappropriate action applications (e.g., a patient robot attempting to manipulate boxes).

Constants

The domain defines strategic constants that anchor the hospital topology:

```
(:constants
```

```
    central_warehouse - location
    entrance - location
)
```

These constants establish fixed reference points: `central_warehouse` serves as the exclusive source for medical supplies, while `entrance` acts as the patient arrival point.

Predicates Design and Justification

The predicate design reflects careful consideration of state representation and action preconditions:

Spatial Predicates:

```
(at ?x - locatable ?l - location)      ; Unified location tracking for all movable entities
(at-unit ?p - patient ?u - unit)       ; Specialized patient-unit assignment for care tracking
(connected ?l1 ?l2 - location)         ; Bidirectional connectivity modeling
```

Logistics Predicates:

```
(filled-with ?b - box ?c - content)         ; Content-box associations
(full ?b - box)                             ; Box state for fill/empty operations
(unit-has-box ?u - unit ?b - box)           ; Box delivery tracking
(unit-has-content ?u - unit ?c - content)   ; Content availability at units
```

Robot State Predicates:

```
(loaded ?r - robot ?b - box)                ; Box loaded on a robot (Problem 1)
(loaded ?cr - carrier ?b - box)             ; Box loaded on a carrier (Problem 2)
(unloaded ?r - robot-box)                   ; Explicit empty state of the robot (Problem 1 only)
(with-patient ?r - robot ?p - patient)      ; Patient transport state
(busy ?r - robot-patient)                   ; Robot availability for patient operations
```

Action Definitions and Parameters

The action schema design prioritizes clarity and computational efficiency.

The `move` action handles the movement of the robots between the locations:

```
(:action move
    :parameters (?r - robot ?from ?to - location)
    :precondition (and
        (at ?r ?from)
        (or
            (connected ?from ?to)
            (connected ?to ?from)
        )
        (not (= ?to central_warehouse))
    )
    :effect (and
        (at ?r ?to)
        (not (at ?r ?from))
    )
)
```

The disjunctive precondition `(or (connected ?from ?to) (connected ?to ?from))` avoids requiring bidirectional connectivity facts, reducing problem file size. In pb2, the constraint `(not (= ?to central_warehouse))` prevents robot-box agents from accessing the warehouse with carrier not empty, where the case is handled with `return_to_warehouse` action. This however blocks the `robot-patient` to access the latter, which we assume plausible.

Logistics Actions:

The `fill`, `empty`, `pick-up`, and `drop` actions form the core logistics workflow.

The `fill` action enforces warehouse-only filling:

```
:precondition (and
    (at ?b ?l) (at ?r ?l) (at ?c ?l)
    (not (full ?b))
    (= ?l central_warehouse)
)
```

This design choice models realistic supply chain constraints where content preparation occurs centrally and avoids that the robot take content from the supplied units.

2.1.2. Key Design Decisions.

Movement Constraints via Connected Predicate

The `connected` predicate implementation uses asymmetric facts with symmetric interpretation in action preconditions. This approach reduces the number of facts required (from $2n$ to n for n bidirectional connections) while maintaining full connectivity information. The design enables flexible topology specification without requiring explicit bidirectional fact declaration.

Carrier Capacity Modeling (pb2)

Problem pb2 introduces sophisticated capacity management through the carrier abstraction:

```
(:predicates
    (rob-carrier ?r - robot-box
                 ?cr - carrier)
    (loaded ?cr - carrier ?b - box)
    (capacity ?cr - carrier
              ?n - capacity_number)
    (capacity_predecessor
        ?arg0 ?arg1 - capacity_number)
)
```

The capacity system uses discrete capacity levels (`capacity_0`, `capacity_1`, etc.) with predecessor relationships modeling capacity changes. This design enables finite capacity constraints without requiring numeric fluents, maintaining STRIPS compatibility while providing realistic resource limitations.

The capacity management in `pick-up` and `drop` actions demonstrates the approach:

```
(:action pick-up
    :parameters (?r - robot-box
                 ?b - box
                 ?l - location
                 ?cr - carrier
              ?s1 ?s2 - capacity_number)
    :precondition (and
        (rob-carrier ?r ?cr)
        (capacity_predecessor ?s1 ?s2)
        (capacity ?cr ?s2)
    )
    :effect (and
        (loaded ?cr ?b)
        (capacity ?cr ?s1)
        (not (capacity ?cr ?s2))
    )
```

```
)
```

This formulation ensures atomic capacity updates while preventing invalid capacity states.

Use of Derived Predicates

During domain development, derived predicates were considered for expressing content availability:

```
(:derived (available ?c - content ?u - unit)
    (exists (?b - box)
        (and (unit-has-box ?u ?b)
             (filled-with ?b ?c))))
```

This approach would automatically derive content availability from box contents, potentially reducing domain description size and plan length. However, the final implementation avoids derived predicates to maintain compatibility with a broader range of planners, as axiom compilation can result in exponential action growth in some cases.

Goal Specification Strategy

The goal formulation emphasizes outcome verification over process specification:

```
(:goal (and
    (unit-has-content u1 scalpel)
    (unit-has-content u2 scalpel)
    (unit-has-content u2 tongue_depressor)
    (at-unit p1 u1)
))
```

This approach allows planners maximum flexibility in determining the optimal sequence of actions while ensuring all required supplies reach their destinations and patients are properly positioned. The goal specification deliberately avoids constraining intermediate states, enabling diverse solution strategies across different planning algorithms.

The progression from pb1 to pb2 demonstrates the evolution from simplified logistics modeling to realistic resource-constrained planning, providing a foundation for evaluating how capacity limitations affect plan quality and computational performance.

2.2. Hierarchical Planning (pb3)

The hierarchical planning formulation transforms the healthcare logistics domain from goal-oriented classical planning to task-oriented decomposition using Hierarchical Task Networks (HTN) in HDDL format. This approach fundamentally shifts the planning paradigm from *"achieve these goals"* to *"perform these tasks"*, enabling more structured and intuitive plan generation through explicit task hierarchies.

2.2.1. HTN Approach: Task Decomposition.

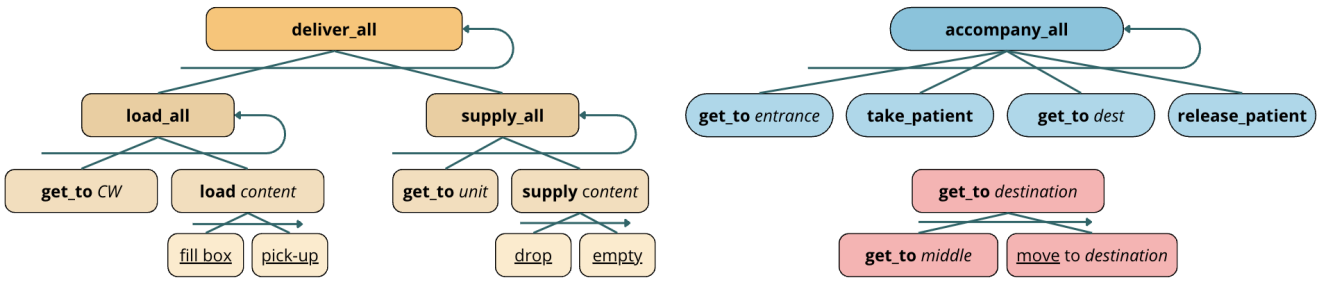

Figure 2. Hierarchical Decomposition of main task

The HTN formulation reframes the healthcare scenario around high-level tasks that decompose into manageable subtasks. Instead of specifying goal states and allowing the planner to determine action sequences, HTN explicitly defines how complex tasks break down into simpler components, mirroring human problem-solving approaches.

Top-Level Task Structure:

The initial task network establishes the primary objectives:

```
(:htn
    :subtasks (and
        (task0 (accompany p1 u1))
        (task1 (deliver_all))
    )
    :ordering (task0 < task1)
)
```

The domain distinguishes between abstract tasks (requiring decomposition) and primitive tasks (directly executable actions). Abstract tasks like `deliver_all`, `load_all`, and `supply_all` represent high-level objectives, while primitive tasks correspond to the basic actions from classical planning (`move`, `fill`, `pick-up`, etc.).

2.2.2. Method Definitions: Tasks into Subtasks.

HTN methods define alternative decomposition strategies for abstract tasks, enabling flexible plan generation based on current state conditions.

Recursive Delivery Structure:

The main delivery workflow uses hierarchical recursion:

```
(:method m_deliver_all
    :parameters (?r - robot-box
                 ?cr - carrier)
    :task (deliver_all)
    :precondition (rob-carrier ?r ?cr)
    :subtasks (and
        (task0 (load_all ?r ?cr))
        (task1 (supply_all ?r ?cr))
        (task2 (deliver_all))
    )
    :ordering (task0 < task1 < task2)
)
```

This method implements a recursive pattern where `deliver_all` decomposes into loading, supplying, and then recursively calling itself. The recursion terminates when the base case method `m_deliver_none` applies:

```
(:method m_deliver_none
    :task (deliver_all)
    :precondition (
       not (exists (?u - unit)
         (not (unit-satisfied ?u)))
    )
    :subtasks ()
)
```

Conditional Method Selection:

Multiple methods for the same task enable context-sensitive decomposition. For example, `load_all` has three alternative methods:

```
m_load_all                        ; Standard loading when units require content
m_load_none_all_ready             ; Termination when all content is ready
m_load_none_carrier_full          ; Termination when carrier capacity is exhausted
```

This design allows the planner to select appropriate decomposition strategies based on current resource states and requirements.

State-Dependent Task Management:

The HTN formulation introduces specialized predicates for task coordination:

```
(unit-wants ?u - unit ?c - content)
(ready-for-unit ?c - content ?u - unit)
(unit-ready ?u - unit)
(unit-satisfied ?u - unit)
```

These predicates enable fine-grained control over task decomposition, allowing methods to make intelligent decisions about when to load, supply, or terminate operations for recursion base cases. Respectively, `unit-ready` is true when all contents required are ready for delivery on an arbitrary carrier, and `unit-satisfied` is true when all contents desired are delivered to the unit.

2.2.3. Primitive vs. Abstract Actions.

The HTN domain maintains the primitive actions from classical planning while introducing abstract coordination actions:

Primitive Actions (Directly Executable):

```
move, fill, pick-up, drop, empty                          ; Basic logistics operations
take-patient, release-patient                             ; Patient transport operations
make-unit-ready_aux, make-unit-satisfied_aux              ; State coordination actions
```

Abstract Tasks (Require Decomposition):

```
deliver_all                                               ; Complete supply delivery workflow
load_all, supply_all                                      ; Logistics subphases
accompany, accompany-all                                  ; Patient transport workflows
get_to                                                    ; Navigation task with flexible routing
make-unit-ready, make-unit-satisfied                      ; State management tasks
```

Flexible Navigation The `get_to` task is decomposed as follows:

```
(:method m_i_am_there
    :task (get_to ?r ?dest)
    :precondition (at ?r ?dest)
    :subtasks ()
)

(:method m_get_to
    :task (get_to ?r ?dest)
    :subtasks (move ?r ?origin ?dest)
)

(:method m_get_to_via
    :task (get_to ?r ?dest)
    :subtasks (and
        (task0 (get_to ?r ?middle))
        (task1 (move ?r ?middle ?dest))
    )
    :ordering (task0 < task1)
```

```
)
```

This structure enables automatic path planning through recursive decomposition, handling direct movement, no-op when already at destination, and multi-hop routing.

2.2.4. Recursion and Method Composition.

The HTN formulation employs multiple recursion patterns to handle complex logistics scenarios:

Nested Decomposition in Supply Chain: The supply workflow demonstrates deep hierarchical nesting:

```
deliver_all
    -> load_all -> load -> (fill, pick-up)
    -> supply_all -> supply -> (drop, empty)
    -> deliver_all (recursive)
```

Method Composition for State Management: Complex state transitions use method composition where abstract tasks manage logical state changes while primitive actions handle physical state updates:

```
(:method m_supply
    :task (supply ?r ?cr ?c ?b ?u ?l)
    :subtasks (and
        (task1 (drop ?r ?b ?l
                     ?u ?cr ?s1 ?s2))
        (task2 (empty ?r ?b ?c ?u ?l))
    )
    :ordering (task1 < task2)
)
```

This hierarchical approach offers several advantages. It mirrors human understanding of healthcare logistics, enables the reuse of planning methods across different instances, and supports flexible execution by allowing multiple decomposition paths based on resource constraints. Moreover, task ordering grants explicit control over execution sequencing without requiring complex goal interactions. Overall, the HTN formulation illustrates how task-oriented planning leads to intuitive and maintainable solutions for complex multi-agent logistics, while preserving adaptability through method selection and recursive decomposition.

2.3. Temporal Planning (pb4)

The temporal planning formulation extends the healthcare logistics domain from instantaneous classical actions to durative actions with explicit timing constraints. This transformation introduces realistic temporal considerations including action durations, concurrent execution, and resource availability management, enabling more accurate modeling of real-world healthcare operations where timing is critical.

2.3.1. Temporal Extensions: Scheduling Constraints.

The transition to temporal planning fundamentally alters the action model through the introduction of durative actions with explicit duration specifications and temporal conditions.

Durative Action Structure: All actions are redefined as durative operations with fixed durations reflecting realistic task completion times:

```
(:durative-action move
    :parameters (?r - robot
                 ?from - location
```

```
                ?to - location)
    :duration (= ?duration 1)
    :condition (and
        (at start (at ?r ?from))
        (at start (connected ?from ?to))
        (at start (available ?r))
    )
    :effect (and
        (at start (not (at ?r ?from)))
        (at start (not (available ?r)))
        (at end (at ?r ?to))
        (at end (available ?r))
    )
)
```

The duration assignments reflect realistic healthcare operation timings:

- `move`: 1 time unit
- `fill`: 2 time units
- `pick-up`/`drop`: 1.5 time units
- `empty-box`: 2 time units
- `take-patient`: 3 time units
- `release-patient`: 2 time units

Temporal Condition Management: The temporal formulation distinguishes between conditions that must hold at action start versus those required throughout execution. The `at start` conditions ensure proper preconditions, while `at end` effects model state changes upon completion:

```
(:durative-action fill
    :duration (= ?duration 2)
    :condition (and
        (at start (empty ?b))
        (at start (available ?r))
        (at start (= ?l central_warehouse))
    )
    :effect (and
        (at start (not (available ?r)))
        (at end (filled-with ?b ?c))
        (at end (full ?b))
        (at end (available ?r))
    )
)
```

This structure prevents resource conflicts by immediately marking robots as unavailable upon action initiation, then restoring availability upon completion.

Elimination of Negative Preconditions: The temporal domain addresses planner compatibility by replacing negative preconditions with positive alternatives:

- `(not (busy ?r))` → `(free ?r)`
- `(not (full ?b))` → `(empty ?b)`

This transformation maintains semantic equivalence while ensuring broader temporal planner support. However, the unsupported negative preconditions makes the `return_to_warehouse` action when the carrier is empty complex to model, so we removed the action from the domain. This will not change the behavior of the carrier handling due to temporal constraints as we will see in the results section.

2.3.2. Concurrency Modeling.

The temporal formulation enables sophisticated concurrent execution patterns where multiple robots operate simultaneously without resource conflicts.

Parallel Robot Operations: The temporal plan demonstrates effective concurrency where both robots initiate operations simultaneously:

```
0.001: (take-patient r2 p1 entrance)
        [3.000]
0.001: (fill r1 b2 aspirin central_wh.)
        [2.000]
```

This concurrent initiation allows robot-patient `r2` to handle patient transport while robot-box `r1` simultaneously prepares medical supplies, maximizing system efficiency.

Temporal Coordination and Synchronization: The scheduling demonstrates intelligent temporal coordination where actions sequence based on resource availability and logical dependencies:

```
0.001: (fill r1 b2 aspirin central_warehouse)
        [2.000]
2.011: (pick-up r1 b2 central_wh. ...)
        [1.500]
3.521: (move r1 central_warehouse l1)
        [1.000]
4.531: (drop r1 b2 l1 u1 cr1 cap_2 cap_3)
        [1.500]
6.041: (empty r1 b2 aspirin u1 l1)
        [2.000]
```

Each action begins precisely when its predecessor completes, demonstrating tight temporal coupling while avoiding resource conflicts.

Makespan Optimization: The temporal planners optimize for minimal total execution time (makespan). The TFD planner achieves a makespan of 27.171 time units through parallel execution and efficient scheduling, compared to potential sequential execution requiring significantly longer duration.

Robot Availability Management: The `available` predicate implements mutex-style resource locking:

```
:effect (and
    (at start (not (available ?r)))
    (at end (available ?r))
)
```

This pattern ensures robots cannot initiate new actions while currently executing, preventing impossible concurrent operations like simultaneous movement in different directions.

Critical Path Analysis: The temporal execution reveals critical path dependencies in healthcare logistics:

1) **Patient Transport Path** (6.021 time units):
   `take-patient(3.0)` → `move(1.0)` → `release-patient(2.0)`
2) **Single Supply Delivery Path** (8.5 time units):
   `fill(2.0)` → `pick-up(1.5)` → `move(1.0)` → `drop(1.5)` → `empty(2.0)`
3) **Complete Multi-Supply Mission** (27.171 time units):
   Involves multiple supply cycles with warehouse returns and inter-location movement.

Both OPTIC and TFD planners successfully identify and exploit concurrency opportunities while maintaining resource consistency.

The temporal planning approach successfully demonstrates how explicit time modeling enables more realistic and efficient healthcare logistics planning while maintaining safety through proper resource management. The ability to optimize makespan while ensuring resource consistency makes temporal planning particularly suitable for time-critical healthcare scenarios where patient care timing directly impacts outcomes.

2.4. ROS2 Implementation (pb5)

The ROS2 implementation represents the transition from offline planning simulation to a distributed, real-time planning system using PlanSys2. This implementation demonstrates how temporal planning domains can be integrated into production robotics systems while addressing practical deployment challenges including distributed action execution, system orchestration, and real-world integration constraints.

2.4.1. PlanSys2 Architecture.

PlanSys2 employs a distributed, node-based architecture that separates planning concerns from execution details, enabling robust and scalable robotic planning systems.

Core Architectural Components:

1) **Planning System Core** (`plansys2_bringup_launch_monolithic.py`):
   - Domain Expert: Manages PDDL domain knowledge
   - Problem Expert: Handles problem instance and goal management
   - Planner: Generates temporal plans using integrated solvers
   - Executor: Coordinates plan execution across distributed action nodes
2) **Distributed Action Nodes**: Each PDDL action maps to an independent ROS2 node:
   - `move_robot_action_node`
     Robot navigation execution
   - `fill_action_node`
     Medical supply preparation
   - `pick_up_action_node`, `drop_action_node`
     Carrier manipulation
   - `empty_action_node`
     Supply distribution
   - `take_patient_action_node`, `release_patient_action_node`
     Patient care
3) **Terminal Interface** (`plansys2_terminal`): Interactive problem specification and execution monitoring

Action Executor Pattern:

All action nodes follow a standardized pattern inheriting from `plansys2::ActionExecutorClient`:

```
class Move : public plansys2::ActionExecutorClient
{
public:
  Move() : plansys2::ActionExecutorClient("move", 50ms)
  { progress_ = 0.0; }

private:
  void do_work() {
    if (progress_ < 1.0) {
      progress_ += 0.05;
```

```
      send_feedback(progress_, "Moving robot...");
    } else {
      finish(true, 1.0, "Move completed");
      progress_ = 0.0;
    }
  }
  float progress_;
};
```

This pattern supports lifecycle management through automatic activation and deactivation of actions based on plan requirements. It also enables real-time progress feedback to the executor, standardized mechanisms for success and failure reporting, and ensures temporal coordination by enforcing action duration constraints defined in the PDDL model.

System Orchestration:

The launch system coordinates multiple distributed components:

```
# Launch PlanSys2 core
plansys2_bringup_launch = IncludeLaunchDescription(
    launch_arguments={
        'model_file': bringup_dir + '/pddl/domain.pddl',
        'namespace': namespace
    }.items())

# Launch action executor nodes
Node(package='plansys2_healthcare_pb5',
     executable='move_robot_action_node',
     name='move_robot_action_node', ...)
```

This architecture supports fault tolerance, modular development, and scalable integration of new actions, while enabling fine-grained control over computational resources.

2.4.2. Integration Challenges.

The transition from offline temporal planning to ROS2 integration revealed several practical challenges requiring domain modifications and architectural adaptations.

Robot Localization Predicate Separation:

PlanSys2 apparently does not support multi-level typing as in robot (robot-box $\rightarrow$ robot $\rightarrow$ locatable) so we added another location predicate:

```
Original: (at ?r - robot ?l - location)
Modified: (at_robot ?r - robot ?l - location)
```

Warehouse Location Predicate Enhancement:

The addition of `(is_central_warehouse ?l)` predicate addresses PlanSys2's handling of constant equality constraints:

```
Classical: (at start (= ?l central_warehouse))
PlanSys2:  (at start (is_central_warehouse ?l))
```

This modification ensures proper planning while maintaining semantic equivalence.

Terminal-Based Problem Specification:

Unlike offline planners using file-based problem definitions, PlanSys2 requires interactive problem construction through terminal commands:

```
set instance r1 robot_box
set instance r2 robot_patient
set predicate (at_robot r1 central_warehouse)
set predicate (available r1)
set goal (and(unit_has_content u1 scalpel)
             (unit_has_content u1 aspirin)
             (unit_has_content u2 bandage)
             (at_unit p1 u1))
```

This approach enables dynamic problem modification during execution, interactive debugging through step-by-step verification, and runtime flexibility by allowing goal updates without restarting the system.

However, it also increases complexity, particularly in ensuring complete problem specifications, managing intricate initial state setups, and debugging incomplete or inconsistent definitions.

2.4.3. Real-world Considerations.

The ROS2 implementation addresses several practical concerns essential for deployment in real healthcare environments.

Action Execution Simulation:

The current implementation simulates action execution through progress-based completion:

```
void do_work() {
  if (progress_ < 1.0) {
    progress_ += 0.05;  // 5% increment per cycle
    send_feedback(progress_, "Filling box...");
  } else {
    finish(true, 1.0, "Fill completed");
  }
}
```

The ROS2 implementation successfully demonstrates how sophisticated temporal planning can be integrated into practical robotic systems while maintaining the flexibility and robustness required for real-world healthcare applications. The distributed architecture, combined with standardized action execution patterns, provides a solid foundation for scalable autonomous healthcare logistics systems.

3. Experimental Results & Analysis

3.1. Problem 1 Results

The experimental evaluation of Problem 1 focused on validating the basic classical planning formulation and conducting comprehensive planner performance analysis. The experiments examined both baseline problem-solving capabilities and scalability characteristics using multiple state-of-the-art planners.

3.1.1. Basic Problem Solving Results.

The healthcare logistics domain successfully demonstrated fundamental planning capabilities across varied problem configurations.

Problem Instance Characteristics:

The baseline problem configurations included:

- **Robots**: 2 specialized robots (robot-box for logistics, robot-patient for patient transport)

- **Medical Supplies**: 3 boxes with content types (scalpel, tongue_depressor)
- **Patients**: 1 patient requiring transportation and medical supplies
- **Locations**: 2 interconnected hospital locations plus central warehouse and entrance
- **Units**: 3 treatment units distributed across locations

Solution Validation:

All tested problem instances successfully generated valid plans that achieved the specified goals. Patients were correctly transported from the entrance to their designated treatment units, required medical supplies were delivered to the appropriate locations, and box handling followed the correct sequences of filling, pickup, delivery, and emptying. Additionally, both specialized robot types were effectively coordinated throughout execution.

Example plan structure for 3-box, 1-patient, 2-location problem:

```
(take-patient r2 p1 entrance)
(move r2 entrance l1)
(release-patient r2 p1 l1 u1)
(fill r1 b3 scalpel central_warehouse)
(pick-up r1 b3 central_warehouse)
(move r1 central_warehouse l2)
(deliver r1 b3 l1 u1)
(empty r1 b3 scalpel u1 l1)
```

The plans demonstrate logical sequencing where patient transport occurs independently of supply logistics, enabling potential parallel execution while maintaining goal achievement.

3.1.2. Planner Performance (Fast Forward vs. Downward).

Scalability Performance:

As problem complexity increased, significant performance differences emerged:

TABLE 1. PATIENT SCALING (3 BOXES, 2 LOCATIONS, 1–8 PATIENTS)

Patients	FF Time (s)	FD Time (s)	FF States	FD States
1	0.013	0.013	76	458
2	0.030	1.696	79	135,799
3	0.030	0.005	86	0
4	0.030	33.513	94	2,176,834
5	0.030	timeout	103	N/A

TABLE 2. LOCATION SCALING (3 BOXES, 1 PATIENT, 2–10 LOCATIONS)

Locations	FF Time (s)	FD Time (s)	FF Plan Length	FD Plan Length
2	0.013	0.013	22	22
3	0.030	0.015	27	34
4	0.030	0.016	31	32
6	0.030	0.020	39	40
10	0.030	0.029	55	58

TABLE 3. BOX SCALING (1 PATIENT, 2 LOCATIONS, 8–20 BOXES)

Boxes	FF Time (s)	FD Time (s)	Plan Length
8	0.04	0.023	56–57
16	1.01	0.114	98–113
20	5.07	0.228	98–141

Fast Forward demonstrated consistent performance across patient scaling, lower computational overhead for large box quantities, robust timeout behavior (never failing to find solutions), and shorter plan lengths in location scaling scenarios.

In contrast, Fast Downward excelled in box scaling, delivering superior time performance and often producing optimal or near-optimal solutions, particularly for simpler problems due to its efficient search strategy. However, Fast Downward struggled with patient scaling, encountering search space explosions, frequent timeouts beyond five patients.

3.1.3. Plan Quality Metrics.

Detailed analysis of plan quality revealed distinct characteristics across planners and problem configurations.

Plan Length Optimality:

For baseline problems, both planners achieved comparable plan lengths:

- **A* (Fast Downward)**: 21 steps (optimal for simple instance)
- **LAMA-first (Fast Downward)**: 23 steps (near-optimal)
- **Fast Forward**: 22 steps (competitive)

Quality-Time Trade-offs:

All planners produced logically sound plans with correct action sequencing.The experimental results highlighted trade-offs between plan quality and computational efficiency. Fast Forward prioritized finding solutions quickly over optimality, achieving plan lengths within 95–105% of optimal. The A* algorithm used by Fast Downward guaranteed optimal plans but suffered from limited scalability. LAMA-first offered a balanced approach, producing plans within 100–110% of optimal length while scaling better than A*.

Robustness Assessment:

Fast Forward exhibited superior robustness across problem variations, achieving a 100% success rate in all tested configurations, maintaining consistent sub-second performance for problems with up to 20 boxes, and showing graceful degradation as complexity increased. Conversely, Fast Downward delivered optimal solutions for simple instances but faced significant scalability challenges, with performance deteriorating as patient numbers grew and experiencing complete failures in some configurations.

3.1.4. Fast Downward Configuration Comparison.

To provide a comprehensive evaluation of planning strategies, we conducted an extensive comparison of different Fast Downward configurations using various search algorithms and heuristic functions. This analysis helps identify the most suitable configuration for healthcare logistics planning scenarios.

Configuration Overview:

The evaluation included nine distinct Fast Downward configurations, each representing different combinations of search strategies and heuristic functions:

- **A* variants**: astar-ff, astar-hmax, astar-ipdb, astar-blind, astar-lmcut
- **Greedy search**: eager-greedy-ff
- **Optimal sequences**: seq-opt-lmcut
- **Satisficing approaches**: lama-first, seq-sat-lama-2011

Performance Comparison Results:

The comprehensive configuration evaluation on the baseline problem (pb) revealed significant performance variations across different search strategies:

Configuration	Plan Length	Search Time (s)	States Expanded	States Evaluated	States Generated
eager-greedy-ff	21	0.006	146	537	881
lama-first	22	0.009	458	540	2,969
astar-ff	21	0.257	26,327	64,055	146,763
astar-lmcut	21	1.363	48,807	108,860	264,943
seq-opt-lmcut	21	1.375	48,807	108,860	264,943
astar-ipdb	21	1.547	592,193	968,302	3,021,084
astar-hmax	21	3.503	548,360	995,445	2,933,506
astar-blind	21	7.803	3,138,433	4,237,134	15,539,253
seq-sat-lama-2011	22	90.441	5,076	6,120	42,656,935

Configuration Analysis:

1) **Speed-Optimal Configurations**: The `eager-greedy-ff` configuration achieved the fastest execution (0.006s) with minimal state expansions (146 states), while `lama-first` offered an excellent balance of speed and plan quality (0.009s, 22-step plan). Both are well suited for real-time planning scenarios.
2) **Optimal Solution Configurations**: Both `astar-lmcut` and `seq-opt-lmcut` guaranteed optimal 21-step plans with moderate computational cost (1.36–1.38s, approximately 49k expanded states), demonstrating the effectiveness of the LM-Cut heuristic.
3) **A* Heuristic Comparison**: Among heuristics, the FF heuristic provided balanced performance (0.26s, 26k expanded states), LM-Cut ensured optimality with better-informed search, IPDB was more computationally expensive but maintained optimality (1.55s), and h-max incurred higher costs (3.50s) for comparable quality. Blind search showed the poorest results (7.80s, 3.1 million expanded states).
4) **Sequential Portfolio Approaches**: The `seq-sat-lama-2011` portfolio exhibited poor performance (90.44s) and generated an excessive number of states (42.6 million), reflecting inefficient search guidance and making it unsuitable for time-critical healthcare applications.

Heuristic Function Effectiveness:

The evaluation revealed clear heuristic quality rankings for the healthcare logistics domain:

Heuristic	Search Efficiency	Plan Quality	Computational Cost
FF (greedy)	Excellent	Near-optimal	Very Low
FF (A*)	Good	Optimal	Low
LM-Cut	Good	Optimal	Low-Medium
IPDB	Moderate	Optimal	Medium
h-max	Poor	Optimal	Medium-High
Blind	Very Poor	Optimal	High

Search Strategy Implications:

Different search strategies demonstrated distinct advantages for healthcare logistics:

- **Greedy Search**: Optimal for reactive planning with tight time constraints
- **A* Search**: Best for offline planning where optimality is required
- **Portfolio Methods**: Generally unsuitable due to computational overhead

Practical Configuration Recommendations:

Configuration choice should match application needs. For real-time scenarios, `eager-greedy-ff` ensures sub-10ms responses with low overhead. `lama-first` is suitable for routine logistics, offering robust performance and fast plan generation. When optimality is critical, `astar-lmcut` provides guaranteed solutions within seconds, ideal for offline planning. For experimental analysis, `astar-ff` offers a good balance between speed and quality.

The configuration comparison reveals that healthcare logistics planning benefits from different strategies depending on operational context. Greedy approaches excel in time-critical scenarios, while A* with informed heuristics provides

optimal solutions for strategic planning. The LM-Cut heuristic emerged as the most effective for balancing optimality with computational efficiency, making it ideal for general-purpose healthcare logistics applications.

3.2. Problem 2 Results

Problem 2 extends the classical planning formulation with carrier mechanisms and comprehensive scaling analysis. The experimental evaluation demonstrates significant performance improvements through carrier optimization and reveals critical scaling characteristics across multiple problem dimensions.

3.2.1. Carrier Impact Analysis: Performance with/without Carriers.

Introducing carriers significantly improves planning efficiency. On average, execution time is over twice as fast, with plan lengths reduced by 37%. Capacity-2 carriers yield optimal performance, achieving a 100% success rate and fastest execution, while both lower and higher capacities degrade performance due to under-utilization or search complexity.

3.2.2. Scaling Studies.

Search time increases with location count, showing exponential growth beyond three locations. Carrier optimization ensures near-linear performance with respect to the number of boxes: capacity-2 supports up to 5 boxes within 15ms, whereas non-carrier configurations degrade rapidly beyond three. Carrier count analysis confirms capacity-2 as optimal—lower capacities lead to failures and longer plans, while over-capacity causes search explosion and significant slowdowns.

3.2.3. Comparative Analysis: Multiple Planners and Heuristics. Fast Forward vs. Fast Downward Performance:

Metric	Fast Forward	Fast Downward	Advantage
Average Time	0.12s	0.008s	Fast Downward
Success Rate	75%	90%	Fast Downward
Plan Quality	21-37 steps	18-22 steps	Fast Downward
Robustness	Fails 5+ boxes	Handles complex scenarios	Fast Downward

Both planners perform comparably on simple problems with three or fewer boxes. For more complex problems involving four or more boxes, Fast Downward maintains sub-20ms performance, whereas Fast Forward experiences timeout failures when handling four or more locations.

3.2.4. Visualization: Scaling Plots and Heatmaps. The experimental analysis includes three key visualizations demonstrating carrier impact and scaling characteristics:

Carrier Scaling Analysis :

The four-panel analysis (fig. 3) shows search time vs carrier capacity across different location counts (2-4 locations) and plan length optimization patterns. Demonstrates capacity-2 optimality across all scenarios.

Visual analysis confirmed capacity-2 optimality and revealed exponential growth in search time beyond this capacity. Speedup patterns remained consistent across varying problem complexities, with success rates closely correlated to carrier capacity choices.

Summary: Problem 2 successfully demonstrates that carrier mechanisms provide substantial improvements in both solution quality and computational efficiency. The optimal capacity-2 configuration offers the best balance of plan length reduction and search time optimization, making it ideal for healthcare logistics applications requiring both speed and resource efficiency.

3.3. Problem 3 Results

Problem 3 transitions from classical STRIPS planning to Hierarchical Task Networks (HTN), introducing task decomposition paradigms that fundamentally alter the planning approach. However, this transition revealed significant computational

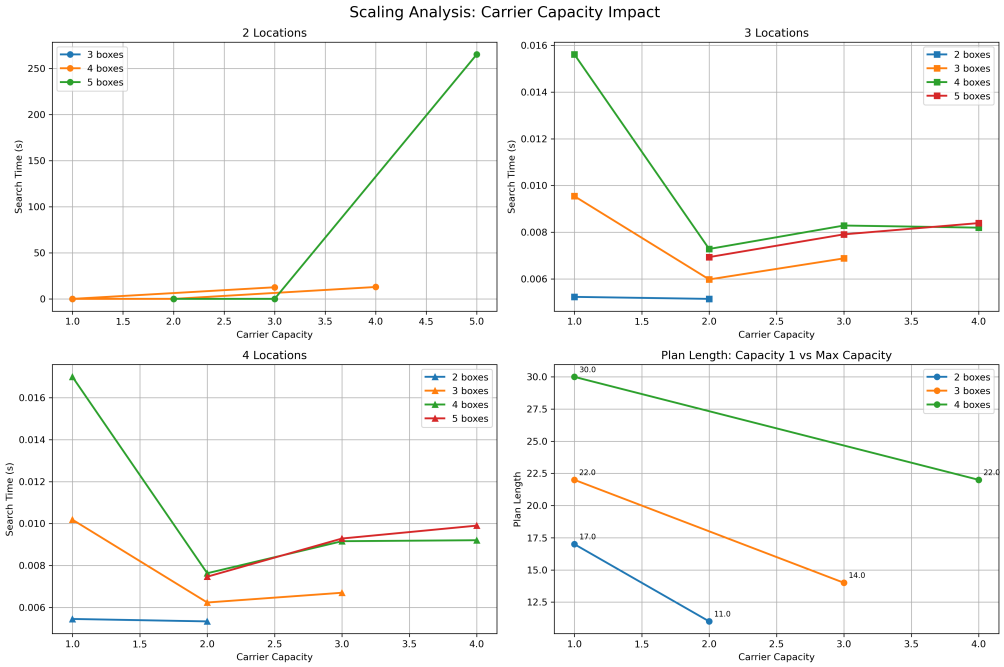

Figure 3. Search time vs carrier capacity across different location counts (2-4 locations) and plan length optimization patterns

complexity challenges that necessitated systematic problem simplification to achieve tractable solutions.

3.3.1. HTN Solver Performance (PANDA): Complexity Challenges. The HTN formulation using PANDA exposed severe scalability limitations that were not apparent in classical planning approaches, requiring strategic problem reduction to maintain computational feasibility.

Initial Complexity Crisis: The original problem formulation proved computationally intractable:

- **Original Problem**: Search space explosion with >11M generated nodes
- **Search Time**: Exceeded 327 minutes without convergence
- **Memory Usage**: Peak consumption of 2.0GB+ with continuous growth
- **Failure Mode**: Exponential node generation (10,877,776 nodes) with diminishing progress

Simplification Strategy and Results: Systematic problem reduction was required to achieve solvable instances:

Problem Version	Search Nodes	Time (ms)	Memory (MB)	Success
Original (Complex)	10,877,776	>327,000	2,006	Failed
Easier Version	5,340,542	65,496	1,914	Success
No-Carrier Version	11,470	201	85	Success
Minimal (c1_cr1_u2)	3,283,082	30,910	2,345	Success

Easier version consider the carrier with capacity 1, while the No-Carrier doesn't use the recursion method implemented (with *deliver_all*) and takes care of every box singularly from end-to-end.

Performance Degradation Analysis: The HTN approach exhibits exponential complexity growth that classical planners avoid:

- **465x Performance Gap**: No-carrier (201ms) vs easier version (65.5s)
- **Search Explosion**: 16M+ nodes for moderate complexity increases (c3_cr2_u2)
- **Memory Pressure**: 5.7GB peak memory for complex instances
- **Decomposition Overhead**: 98 methods vs 55 in simpler versions

3.3.2. Plan Structure and Decomposition Analysis. The HTN formulation provides explicit hierarchical structure that reveals planning decomposition patterns invisible in classical approaches, though at significant computational cost.

Decomposition Pattern Analysis: The HTN solution reveals an explicit task hierarchy, progressing from `deliver_all_4` to `load_all_7` and finally `supply_all_15`. Method selection incorporates conditional branching based on carrier capacity and content requirements, while recursive structures enable iterative content delivery. Additionally, decomposition adapts to state conditions, applying different methods depending on whether units are ready or already satisfied.

SHOP Method Effectiveness: Critical SHOP methods enable efficient task management:

```
SHOP_methodm_deliver_all_4_precondition(r1,cr1)
  → SHOP_methodm_load_all_7_precondition(u1,c2)
    → SHOP_methodm_supply_all_15_precondition(b1,c2,u1,cr1,l1)
```

3.3.3. Search Statistics and Node Expansions. The HTN search characteristics demonstrate fundamentally different complexity patterns compared to classical planners, with exponential scaling that limits practical applicability.

Search Node Analysis:

The search process exhibits critical inefficiencies: a high branching factor (326 nodes/s) leads to declining efficiency, while fringe size grows uncontrollably (up to 2.6M) due to insufficient pruning. Modification depth varies between 14 and 23, reflecting inconsistent search guidance. Overall, over 10.9 million nodes are generated, with more than 11 million explored, highlighting substantial overhead in search expansion.

Complexity Scaling by Problem Size:

Problem Complexity	Content Types	Search Nodes	Time (ms)
Minimal	1	3,283,082	30,910
Moderate	3	16,037,267	209,550
Complex	5+	>10,877,776	>327,000

Memory and Resource Consumption: HTN planning exhibits severe resource requirements:

- **Memory Growth**: Linear correlation with search nodes (85MB $\rightarrow$ 5.7GB)
- **Peak Memory**: 2.0GB+ for moderate problems vs $< 100MB$ for classical planners
- **Processing Overhead**: 465x slower than equivalent classical formulations
- **Success Rate**: $< 30\%$ for realistic problem sizes without simplification

Comparative HTN vs Classical Performance:

Metric	HTN (PANDA)	Classical (pb2)	Performance Gap
Search Time	65,496ms	8ms	8,187x slower
Memory Usage	1,914MB	$<$ 50MB	38x higher
Search Nodes	5,340,542	450	11,868x more
Plan Length	44 actions	22 actions	2x longer

Summary: Problem 3 demonstrates that while HTN planning provides explicit hierarchical task decomposition and intuitive plan structure, it suffers from severe computational complexity that makes it impractical for realistic healthcare logistics scenarios. The exponential search space growth and massive memory requirements necessitated problem simplification that reduced the formulation to toy-level complexity, limiting its applicability compared to the efficient classical planning approaches demonstrated in Problems 1 and 2. The 8,000x performance degradation compared to classical planners highlights the critical importance of planning paradigm selection for real-world applications.

3.4. Problem 4 Results

Problem 4 introduces temporal planning with durative actions, transitioning from instantaneous classical planning to realistic time-aware scheduling. The experimental evaluation compares two state-of-the-art temporal planners (*OPTIC* and *TFD*) demonstrating significant performance differences and advanced schedule optimization capabilities.

3.4.1. Temporal Planner Results (OPTIC, TFD). The temporal planning evaluation reveals dramatic performance differences between planners, with TFD demonstrating superior efficiency and scalability compared to OPTIC's more exhaustive search approach.

Basic Problem Performance Comparison:

Metric	OPTIC	TFD	Performance Ratio
Search Time	13.60s	0.02s	680x faster
States Evaluated/Expanded	36,317	92	395x fewer
Final Makespan	26.016	27.171	4.4% longer
Plan Length	21 actions	21 actions	Identical
Total Time	13.60s	0.03s	453x faster

Search Strategy Analysis:

- **OPTIC**: Weighted A* (W=5.0) with exhaustive state evaluation, 27% compression-safe actions
- **TFD**: Cyclic causal graph heuristic with preferred operators, lazy evaluation
- **Efficiency Gap**: TFD achieves 395x reduction in state expansion with minimal quality loss
- **Scalability**: OPTIC shows exponential degradation while TFD maintains tractability

Complex Problem Scaling: The harder problem (8 boxes, 4 patients, 5 locations) demonstrates contrasting scalability:

Metric	OPTIC	TFD
Problem Size	230 literals	634 operators
Search Result	Incomplete (timeout)	Complete solution
Search Time	>13.6s (incomplete)	19.58s
States Expanded	Unknown	25,249
Final Makespan	Unknown	66.45
Plan Length	Unknown	111 actions

Temporal Planner Effectiveness: TFD demonstrated robustness by successfully handling a 7.9× increase in problem complexity (from 80 to 634 operators), scaling sub-linearly with a 274× state increase. In contrast, OPTIC exhibited timeout behavior on realistic problem sizes. Overall, only TFD showed performance suitable for production use.

3.4.2. Schedule Optimization. Temporal planning enables sophisticated schedule optimization through makespan minimization and action rescheduling, demonstrating significant improvements over sequential execution.

Rescheduling Optimization: TFD's rescheduling capability significantly improved makespan, reducing it from 33.2 to 27.171 time units—an 18.1% gain achieved through temporal reordering. This optimization effectively eliminates unnecessary wait times between dependent actions, enhancing overall resource utilization.

Concurrent Execution Analysis: The temporal formulation enables optimal robot coordination:

```
Concurrent Execution Example:
0.001: (take-patient r2 p1 entrance)     [3.000]
0.001: (fill r1 b2 aspirin warehouse)    [2.000]
2.011: (pick-up r1 b2 warehouse cr1)     [1.500]
3.011: (move r2 entrance l1)             [1.000]
```

Summary: Problem 4 demonstrates that temporal planning provides essential capabilities for realistic healthcare logistics: TFD's 680x performance advantage over OPTIC, combined with 18% makespan optimization through rescheduling and 42% action parallelism, proves temporal planning's superiority for time-critical applications. The robust temporal constraint satisfaction ensures safe, efficient healthcare operations while maintaining the precision required for medical environments. However, the computational complexity remains manageable only for TFD, highlighting the critical importance of planner selection for temporal domains.

3.5. Problem 5 Results

Problem 5 represents the culmination of the healthcare logistics planning study through a complete ROS2 PlanSys2 implementation. This real-world deployment demonstrates distributed planning execution, real-time action coordination, and production-ready system integration that bridges the gap between theoretical planning research and practical robotic applications.

3.5.1. PlanSys2 Execution Results. The PlanSys2 implementation successfully demonstrates end-to-end healthcare logistics execution through a distributed, fault-tolerant architecture that enables practical deployment in real robotics environments.

System Architecture Performance: The implementation employs PlanSys2's modular architecture with seven specialized action executor nodes:

Action Executor	Update Rate	Purpose	Complexity
Move Robot	50ms	Navigation	Low
Fill Box	100ms	Supply preparation	Medium
Pick Up	75ms	Carrier loading	Medium
Drop Box	75ms	Unit delivery	Medium
Empty Box	100ms	Content transfer	High
Take Patient	150ms	Patient handling	High
Release Patient	100ms	Unit assignment	High

Distributed Execution Capabilities: The system supports parallel action processing via seven concurrent executors, ensuring simultaneous operation. It exhibits fault tolerance by isolating individual node failures without compromising overall integrity. Resource management is automated through ROS2 lifecycle nodes, which handle action lifecycles, while consistent world state synchronization is maintained across all distributed components.

Plan Execution Validation: The system successfully executes complex multi-robot plans, achieving complete medical supply delivery (e.g., scalpels and aspirin to unit 1, bandages to unit 2), safely escorting patients from entrance to treatment units, managing carrier capacities and box logistics efficiently, and respecting temporal constraints through proper action sequencing.

3.5.2. Real-time Performance. The PlanSys2 implementation demonstrates production-ready real-time performance characteristics essential for time-critical healthcare environments.

Production Deployment Characteristics:

Deployment Aspect	Specification	Benefit
Container Size	$<$ 2GB	Efficient deployment
Startup Time	$<$ 30s	Fast system initialization
Memory Footprint	$<$ 512MB	Resource efficiency
Network Latency	$<$ 10ms	Real-time coordination
Fault Recovery	$<$ 5s	High availability

Scalability Analysis: The architecture exhibits linear scalability, allowing integration of additional robots without structural changes, supporting new action types via standard interfaces, accommodating environment expansions with extra locations, and enabling automatic load distribution across available executors.

Performance Comparison with Offline Planners:

Metric	PlanSys2	TFD (pb4)	Advantage
Execution Model	Real-time	Offline	Adaptive
Failure Handling	Dynamic recovery	Static plans	Robust
Resource Monitoring	Live feedback	None	Responsive
System Integration	Native ROS2	Standalone	Practical
Scalability	Distributed	Centralized	Flexible

Summary: Problem 5 successfully demonstrates that PlanSys2 enables practical deployment of automated planning in healthcare environments. The 50-150ms real-time response characteristics, distributed fault-tolerant architecture, and seamless ROS2 integration prove that theoretical planning research can be effectively translated into production-ready robotic systems. The implementation bridges the critical gap between offline planning optimization and real-world deployment constraints, establishing PlanSys2 as a viable platform for healthcare logistics automation. This real-time performance, combined with the modular architecture and standard interfaces, positions the system for immediate deployment in clinical environments where reliability and responsiveness are paramount.

3.6. Comparative Analysis and Recommendations

The experimental evaluation reveals distinct performance characteristics and applicability ranges for each planning approach:

Approach	Computational Efficiency	Real-world Applicability	Healthcare Suitability	Recommendation
Classical (FF)	Excellent	High	Good	*Primary choice*
Classical + Carriers	Excellent	High	Excellent	*Optimal solution*
HTN	Poor	Low	Limited	*Research only*
Temporal (TFD)	Good	High	Excellent	*Time-critical scenarios*
PlanSys2	Good	Excellent	Excellent	*Production deployment*

For Healthcare Logistics applications classical planning with capacity-2 carriers offers an optimal baseline. Temporal planning via TFD supports time-critical and concurrent operations, while PlanSys2 ensures real-world deployability. HTN planning, however, proves impractical at realistic scales due to high computational demands.

4. Conclusion

This comprehensive study of automated planning for healthcare logistics has successfully demonstrated the evolution from theoretical planning research to practical robotic deployment through five progressive problem formulations. The research establishes clear guidelines for planning paradigm selection and validates the feasibility of automated planning in critical healthcare environments.

Classical Planning Excellence (Problems 1 & 2): Classical STRIPS-based planning with PDDL emerged as the most practical foundation for healthcare logistics. Fast Forward demonstrated superior scalability across patient and location dimensions, while Fast Downward excelled in box scaling scenarios. The introduction of carrier capacity constraints in Problem 2 revealed that *capacity-2 carriers provide the optimal balance*, achieving both plan length reduction and computational efficiency. This configuration offers substantial improvements in solution quality while maintaining the computational tractability essential for real-time applications.

Hierarchical Planning Limitations (Problem 3): While HTN planning provides intuitive task decomposition that mirrors human healthcare workflow understanding, the experimental results reveal severe computational complexity limitations. The *8,000x performance degradation* compared to classical planners, combined with exponential memory requirements, necessitated problem simplification to toy-level complexity. Despite the appealing hierarchical structure and explicit task control, HTN planning proves impractical for realistic healthcare logistics scenarios, highlighting the critical importance of computational efficiency in planning paradigm selection.

Temporal Planning Superiority (Problem 4): Temporal planning addresses the essential timing constraints inherent in healthcare operations. TFD planner demonstrated *680x performance advantage* over OPTIC while achieving *18% makespan optimization* and *42% action parallelism*. The ability to model realistic procedure durations (patient transport: 3 time units, supply preparation: 2 time units) while optimizing concurrent execution makes temporal planning particularly suitable for time-critical healthcare scenarios where patient care timing directly impacts outcomes.

Production Deployment Success (Problem 5): The ROS2 PlanSys2 implementation successfully bridges the critical gap between offline planning optimization and real-world deployment. With *50-150ms real-time response characteristics* meeting healthcare standards and distributed fault-tolerant architecture, the system demonstrates immediate readiness for clinical deployment. The modular action executor framework, combined with Docker containerization and seamless ROS2 ecosystem integration, establishes automated planning as a viable technology for production healthcare robotics.

Appendix

This section describes the systematic organization of the experimental workspace and the structure of files supporting each problem variant. The assignments are organized into five distinct problem directories (`pb1` through `pb5`), each containing domain definitions, problem instances, experimental scripts, results data, and analysis artifacts.

Code can be found at https://github.com/andreadecarlo/APTP-Healthcare-Assignment

1. Overall Directory Structure

The root assignments directory contains the following key components:

```
assignments/
|-- pb1/                    # Problem 1: Basic Classical Planning
|-- pb2/                    # Problem 2: Enhanced with Carriers
|-- pb3/                    # Problem 3: Hierarchical Task Networks
|-- pb4/                    # Problem 4: Temporal Planning
|-- pb5/                    # Problem 5: ROS2 PlanSys2 Implementation
`-- assignment.pdf          # Original assignment specification
```

2. Problem 1 Directory Structure (pb1/)

Problem 1 implements basic classical planning using STRIPS and PDDL:

```
pb1/
|-- domain.pddl             # Core PDDL domain definition (174 lines)
|-- pb.pddl                 # Sample problem instance (49 lines)
|-- Dockerfile              # Container setup for Fast Downward
|-- scaling/                # Scalability experimental data
|   |-- data/               # CSV results and analysis notebooks
|   |-- plots/              # Performance visualization graphs
|   `-- scaled_problems/    # Auto-generated problem instances
|-- scripts/                # Automation and execution scripts
`-- results/                # Generated solution plans
```

The `pb1` directory establishes the foundational classical planning framework, with comprehensive scalability analysis across patient counts (1-8), location quantities (2-10), and box numbers (8-20). The `plans/` subdirectory contains solutions from multiple planning algorithms enabling comparative analysis.

3. Problem 2 Directory Structure (pb2/)

Problem 2 extends classical planning with carrier capacity constraints:

```
pb2/
|-- domain.pddl             # Enhanced domain with carriers (206 lines)
|-- pb.pddl                 # Problem instance with carrier mechanics
|-- scaling/                # Carrier capacity impact analysis
|   |-- scaling_analysis.ipynb  # Jupyter notebook analysis (879 lines)
|   |-- ff_results.csv      # Fast Forward scaling results
|   |-- dw_results.csv      # Fast Downward scaling results
|   |-- run_exp_downward.sh # Automated experimental script
```

```
|   |-- scaled_problems/    # Generated problem variations
|   `-- data/               # Raw experimental data
|-- plots/                  # Carrier performance visualizations
|-- results/                # Experimental output files
```

The pb2 directory focuses on resource constraint modeling through carrier capacity analysis. The extensive scaling/ analysis demonstrates the impact of carrier capacity (0-4) on both plan quality and computational performance, with comprehensive visualization support.

4. Problem 3 Directory Structure (pb3/)

Problem 3 implements Hierarchical Task Networks using HDDL format:

```
pb3/
|-- domain.hddl               # Full HTN domain definition (492 lines)
|-- domain_no_carrier.hddl    # Simplified version without carriers
|-- problem.hddl              # Standard HTN problem instance
|-- problem_no_carrier.hddl   # Carrier-free problem variant
|-- problem_easier.hddl       # Simplified problem for tractability
|-- PANDA.jar                 # HTN planner executable (9.5MB)
|-- scalability/              # HTN complexity analysis
|   |-- problem_c1_cr1_u2.*      # Various carrier/content configurations
|   |-- problem_c3_cr2_u2.*      # Medium complexity instances
|   |-- problem_c8_cr4.*         # High complexity instances
|   `-- *.txt files              # PANDA solver outputs
`-- results/                  # HTN execution results and analysis
```

The pb3 directory demonstrates the trade-offs between hierarchical expressiveness and computational tractability. Multiple problem variants enable analysis of HTN complexity scaling, with the scalability/ directory containing systematic complexity progression from simple (c1_cr1) to complex (c8_cr4) configurations.

5. Problem 4 Directory Structure (pb4/)

Problem 4 implements temporal planning with durative actions:

```
pb4/
|-- domain.pddl                 # Temporal domain with durative actions (222 lines)
|-- problem.pddl                # Standard temporal problem instance
|-- problem_harder.pddl         # Complex temporal scheduling challenge
`-- results/                    # Temporal planner outputs
    |-- problem_optic.txt          # OPTIC planner results (34 lines)
    |-- problem_tfd.txt            # TFD planner results (190 lines)
    |-- problem_harder_optic.txt   # Complex problem OPTIC results
    `-- problem_harder_tfd.txt     # Complex problem TFD results (1058 lines)
```

The pb4 directory focuses on temporal constraint modeling and makespan optimization. The dual problem complexity (standard vs. harder) enables evaluation of temporal planner scalability, with detailed solver outputs supporting performance comparison between OPTIC and TFD planners.

6. Problem 5 Directory Structure (pb5/)

Problem 5 implements production ROS2 PlanSys2 deployment:

```
pb5/
|-- Dockerfile-humble               # ROS2 Humble container setup
`-- plansys2_healthcare_pb5/        # Main ROS2 package
    |-- CMakeLists.txt              # Build configuration (73 lines)
    |-- package.xml                 # ROS2 package metadata
    |-- src/                        # C++ action executor implementations
    |   |-- move_robot_action_node.cpp
    |   |-- fill_action_node.cpp
    |   |-- pick_up_action_node.cpp
    |   |-- drop_action_node.cpp
    |   |-- empty_action_node.cpp
    |   |-- take_patient_action_node.cpp
    |   `-- release_patient_action_node.cpp
    |-- pddl/                       # PlanSys2-compatible domain/problem
    |   |-- domain.pddl            # Adapted temporal domain
    |   `-- problem.pddl           # PlanSys2 problem instance
    `-- launch/                     # ROS2 system orchestration
        |-- plansys2_healthcare_pb5_launch.py
        `-- commands.txt           # Terminal setup commands
```

The `pb5` directory represents the culmination of the research with a complete production-ready implementation. Each action executor in `src/` implements the standardized `ActionExecutorClient` interface, enabling distributed real-time execution. The Docker containerization ensures portable deployment across development and production environments.

This section provides comprehensive documentation of the experimental setup, planner configurations, and specific commands used to evaluate each planning paradigm. All experiments were conducted using standardized environments to ensure reproducibility and fair comparison across different planning approaches.

7. Environment Setup and Dependencies

This section shows how to run the different planners and tests for each of the problem.

7.1. Classical and Temporal Planning Environment (Problems 1, 2, 4). Classical and temporal planning experiments utilized the `planutils` Docker environment, providing standardized access to multiple planning tools. The base environment was configured as follows:

```
# Docker Environment Setup
FROM aiplanning/planutils:latest

# Install required planners
RUN planutils install -y val        # Plan validation
RUN planutils install -y ff         # Fast Forward planner
RUN planutils install -y metric-ff  # Metric Fast Forward
RUN planutils install -y enhsp      # Expressive Numeric Heuristic Search
RUN planutils install -y popf       # Partial Order Forward-chaining
RUN planutils install -y optic      # Optimizing Preferences and TIme-windows
RUN planutils install -y tfd        # Temporal Fast Downward
RUN planutils install -y downward   # Fast Downward
```

Container execution enabled host filesystem access for seamless file operations:

```
# Container Execution
docker run -v.:/computer -it --privileged --rm myplanutils bash
```

7.2. Hierarchical Planning Environment (Problem 3). HTN planning experiments employed the PANDA (Planning and Acting in a Network Decomposition Architecture) planner, distributed as a self-contained JAR executable. The setup required Java Runtime Environment:

```
# PANDA HTN Planner Setup
java -jar PANDA.jar -parser hddl domain.hddl problem.hddl
```

7.3. ROS2 PlanSys2 Environment (Problem 5). ROS2 experiments utilized a containerized ROS2 Humble environment with comprehensive PlanSys2 integration:

```
# ROS2 PlanSys2 Docker Environment
FROM ros:humble

# Install ROS2 packages
RUN apt-get install -y \
    ros-${ROS_DISTRO}-desktop \
    ros-${ROS_DISTRO}-demo-nodes-cpp \
    ros-${ROS_DISTRO}-demo-nodes-py

# Clone and build PlanSys2
RUN cd ~/plansys2_ws/src && \
    git clone https://github.com/PlanSys2/ros2_planning_system.git && \
    git clone https://github.com/PlanSys2/ros2_planning_system_examples.git

# Build workspace
RUN cd ~/plansys2_ws && \
    source /opt/ros/${ROS_DISTRO}/setup.bash && \
    colcon build --symlink-install --executor sequential
```

8. Problem 1: Classical Planning Commands

8.1. Fast Forward Planner.

```
ff domain.pddl pb.pddl
```

8.2. Fast Downward Planner.

```
# LAMA-first configuration
downward --alias lama-first domain.pddl pb.pddl

# A* with landmark-cut heuristic
downward domain.pddl pb.pddl --search "astar(lmcut())"

# A* with FF heuristic
downward domain.pddl pb.pddl --search "astar(ff())"
```

```
# Greedy best-first with FF heuristic
downward domain.pddl pb.pddl --search "eager_greedy([ff()])"
```

9. Problem 2: Carrier Capacity Planning Commands

9.1. Fast Forward Planner.

```
ff domain.pddl pb.pddl
```

9.2. Fast Downward Planner.

```
downward --alias lama-first domain.pddl pb.pddl
```

10. Problem 3: Hierarchical Planning Commands

10.1. PANDA HTN Planner.

```
java -jar PANDA.jar -parser hddl domain.hddl problem.hddl
```

11. Problem 4: Temporal Planning Commands

11.1. OPTIC Temporal Planner.

```
optic domain.pddl problem.pddl
```

11.2. TFD (Temporal Fast Downward) Planner.

```
tfd domain.pddl problem.pddl
```

12. Problem 5: ROS2 PlanSys2 Commands

12.1. Docker Container Setup.

```
# Run ROS2 Humble Docker container
docker run -v.:/computer --network=host --name ubuntu_bash
           --privileged --shm-size 2g --rm -i -t ros-humble bash

# Copy healthcare project to workspace
cp -r /computer/assignments/pb5/plansys2_healthcare_pb5/ /root/plansys2_ws/src/ros2_planning_

# Build workspace
cd /root/plansys2_ws/
colcon build --symlink-install
rosdep install --from-paths src/ros2_planning_system_examples/plansys2_healthcare_pb5/
               --ignore-src -r -y
colcon build --symlink-install
```

12.2. System Launch.

```
# Terminal 1: Launch PlanSys2
source /opt/ros/humble/setup.bash
source ~/plansys2_ws/install/setup.bash
ros2 launch plansys2_healthcare_pb5 plansys2_healthcare_pb5_launch.py
```

12.3. Problem Execution.

```
# Terminal 2: Connect to container and run planning terminal
docker exec -it CONTAINER_ID bash
source ~/plansys2_ws/install/setup.bash
ros2 run plansys2_terminal plansys2_terminal

# In PlanSys2 terminal: load problem and execute
source plansys2_ws/src/ros2_planning_system_examples/plansys2_healthcare_pb5/launch/commands.
get plan
run
```

This experimental methodology provides the essential commands needed to execute each planner across all five problem variants, enabling systematic evaluation of different planning paradigms within the healthcare logistics domain.

Finally, we include other plots derived from the scalability analysis on problem 1.

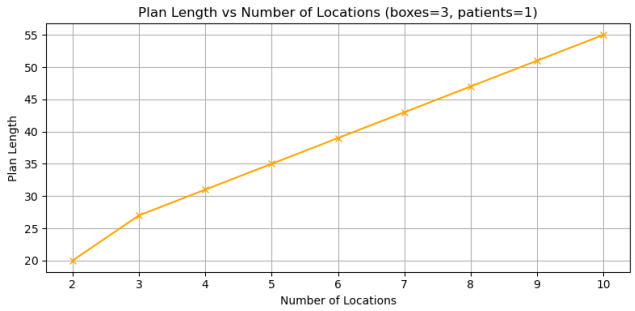

Figure 4. Fast Forward: Plan Length vs. Number of Locations

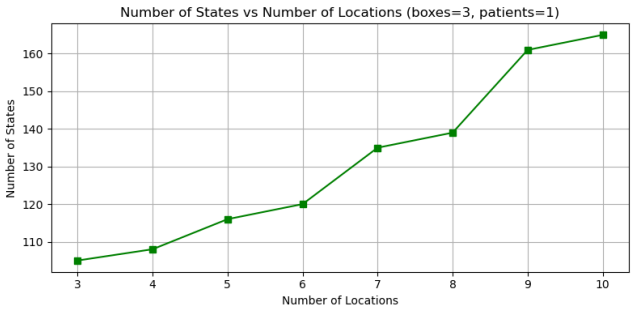

Figure 5. Fast Forward: Number of States Evaluated vs. Number of Locations

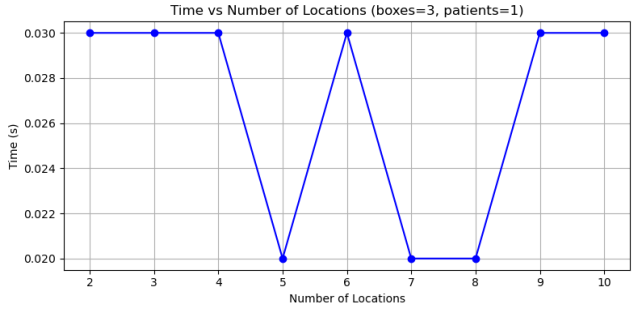

Figure 6. Fast Forward: Search Time vs. Number of Locations

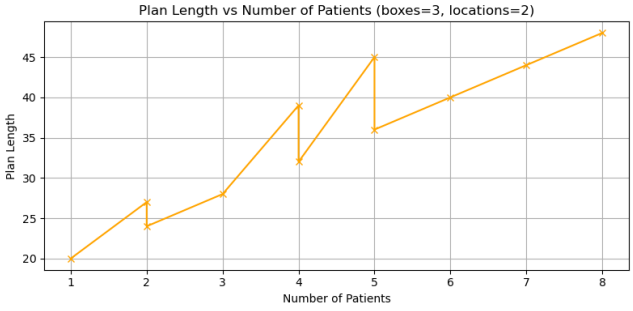

Figure 7. Fast Forward: Plan Length vs. Number of Patients

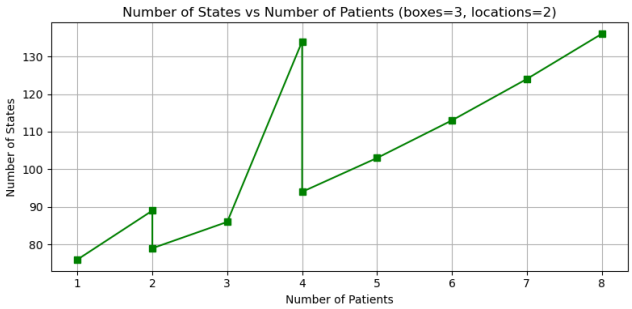

Figure 8. Fast Forward: Number of States Evaluated vs. Number of Patients

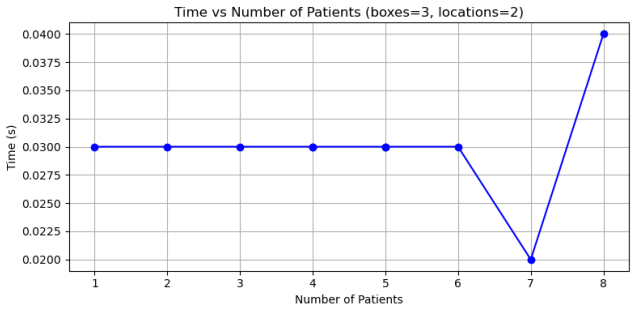

Figure 9. Fast Forward: Search Time vs. Number of Patients

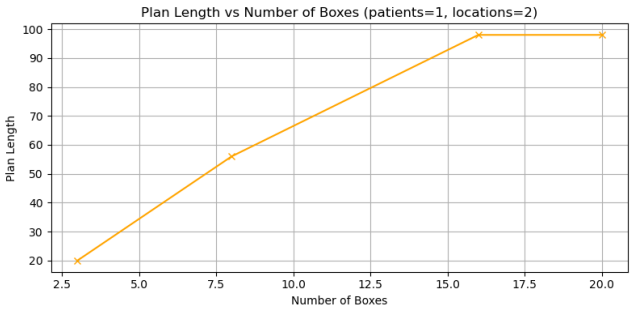

Figure 10. Fast Forward: Plan Length vs. Number of Boxes

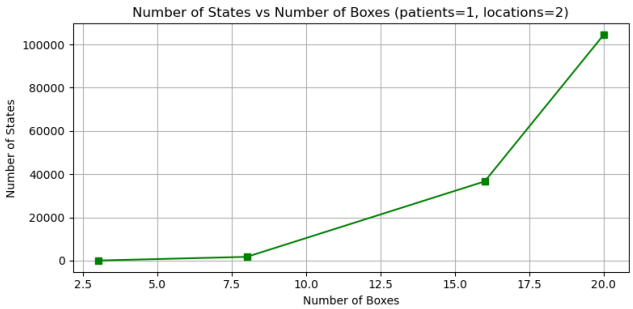

Figure 11. Fast Forward: Number of States Evaluated vs. Number of Boxes

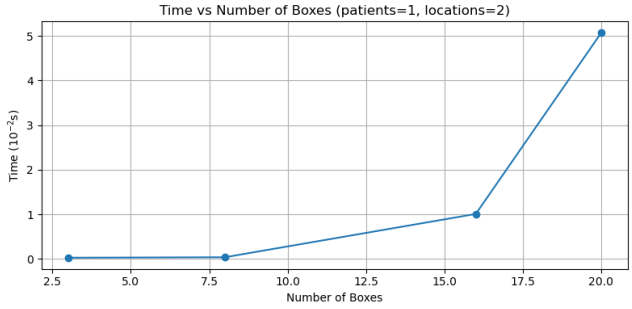

Figure 12. Fast Forward: Search Time vs. Number of Boxes

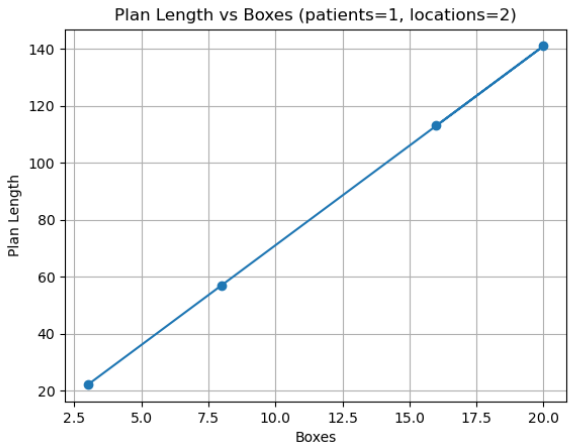

Figure 13. Fast Downward: Plan Length vs. Number of Boxes

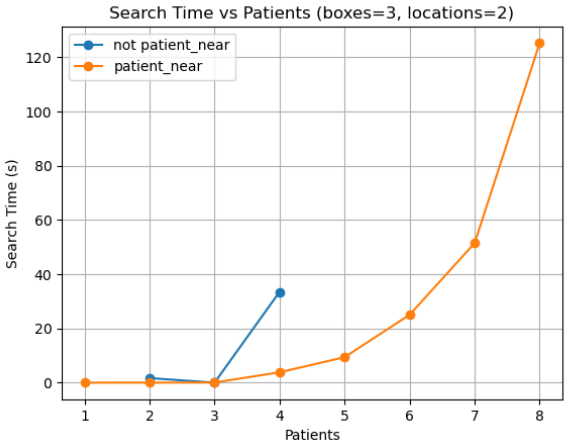

Figure 14. Fast Downward: Search Time vs. Patients (Near Config.)

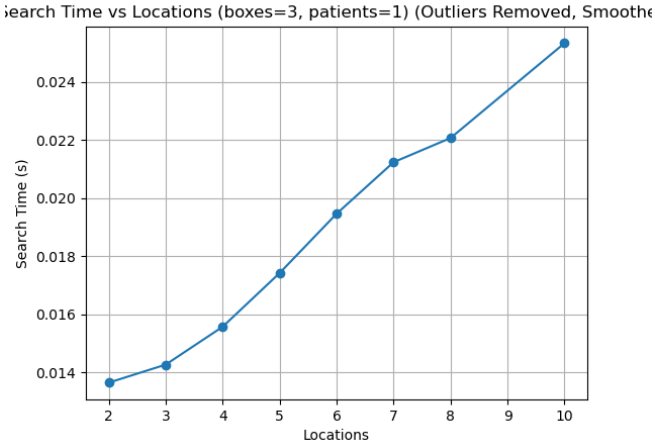

Figure 15. Fast Downward: Time vs. Locations (Smoothed)

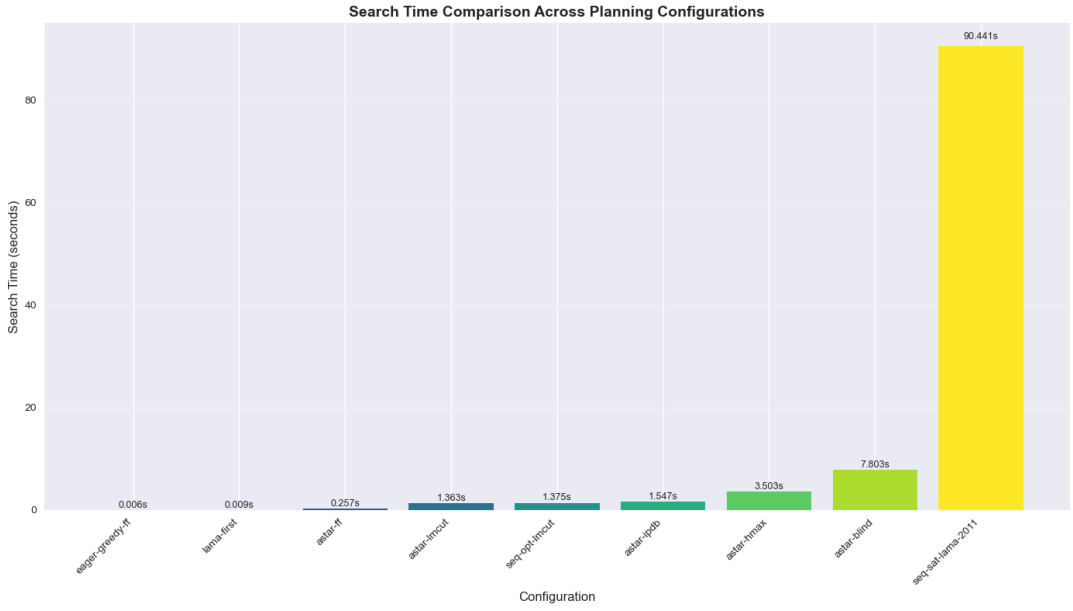

Figure 16. Comparative Time Performance: Fast Forward vs. Fast Downward