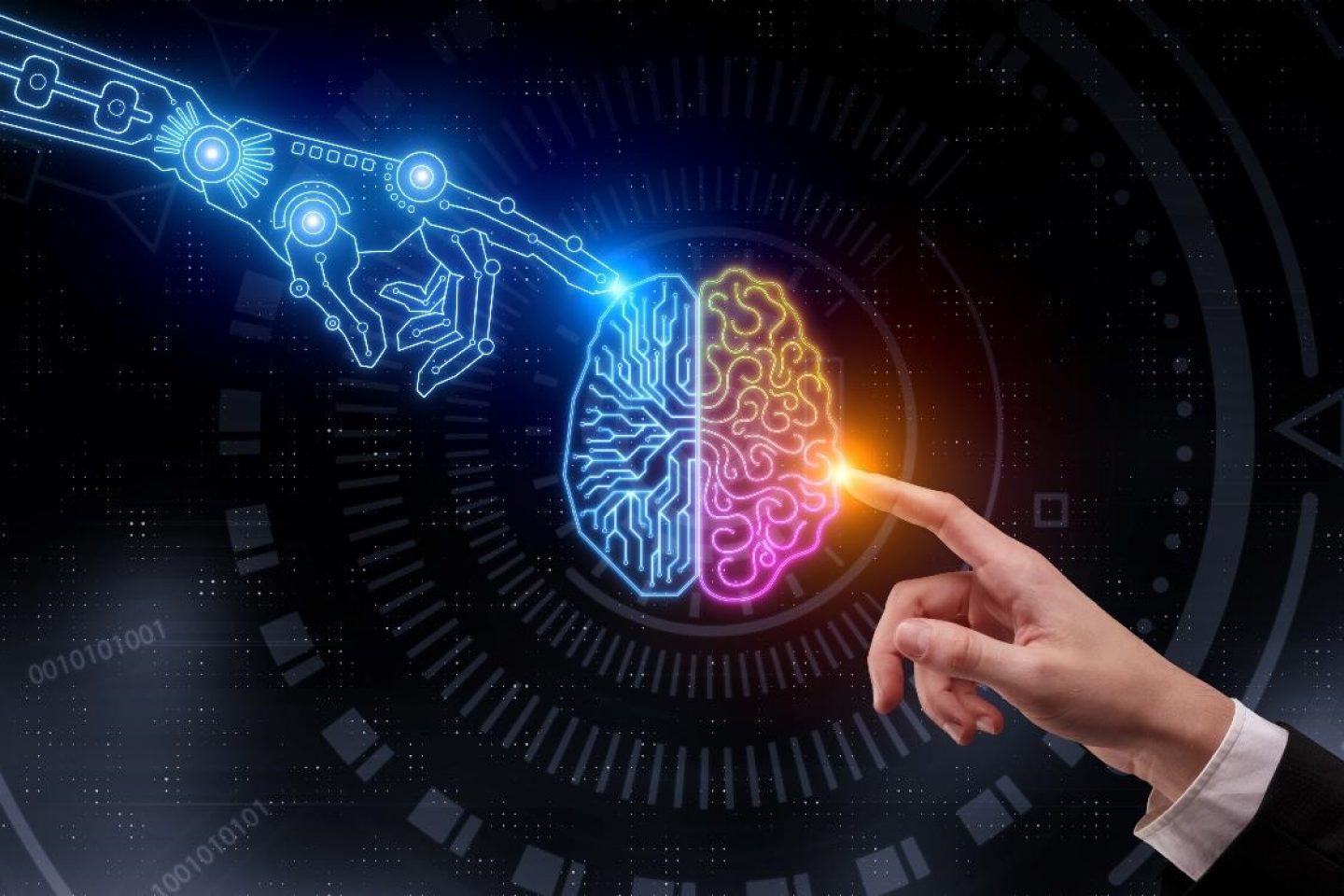
Artificial Intelligence project



**Table of Contents:**

1. Algorithm Background
   1. Logical Agents
      1. Knowledge Based Agents
      2. Logic Representation, Reasoning, and Propositional Logic.
      3. Propositional Theorem Proving
      4. First Order Logics
   2. Planning
      1. Algorithms for Planning
2. Introducing the Problem
   1. Introducing the Problem
   2. The Domain
      1. The Requirements
      2. The Predicates
      3. The Actions
   3. The Problem
      1. Objects
      2. Initialization
      3. The Goals
3. Planner Implementation
   1. The Simple Case Output
   2. More Cases and Analysis.
   3. Conclusions

**1.** **Algorithm Background**

**1.1**  **Logical Agents**

Humans, it seems, know things; and what they know helps them do things.These are not empty statements. They make strong claims about how the intelligence of humans is achieved—not by purely reﬂex mechanisms but by processes of reasoning that operate on internal representations of knowledge. In AI, this approach to intelligence is embodied in knowledge-based agents.

* + 1. **Knowledge Based Agents:**

The central component of a knowledge-based agent is its knowledge base, or KB. A knowledge base is a set of sentences.

Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world, when a sentence is not derived by other sentences, we call this sentence – axiom. There is a way of adding new knowledge into the KB, these operations are called “ask” and “tell”, both operations may involve **inference** – that is, deriving new sentences from old. Inference myst obey the requirement that when one ASKS a question of the KB, the answer should follow from what has been told(“tell”) to the KB previosely.

Another important aspect in initialing the KB, is **background knowledge**.

Each time the agent program is called, it does three things. First, it “tells” the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and the agent executes the action.

This whole procedure executes with the help of three functions;

* Make percept sentence – constructs a sentence asserting percept is perceived at given time.
* Make action query – constructs a sentence that asks what action is needed to be done
* Make action sentence – constructs a sentence asserting that the chosen action was executed.

The KB is not an arbitrary program, it is amenable to a description at the **knowledge level**, where we need to specify only what the agent knows and what its goals are. Notice that the knowledge level is independent, and the implementation level, which can gain knowledge from a variety of sources.

A **declarative** approach is taken when constructing a KB agent, the agent designer “tells” the KB sentences until the agent knows how to operate in its environment, in contrast to the **procedural** approach which encodes desired behaviors directly as program code.

* + 1. **Logic Representation, Reasoning, and Propositional Logic.**

In the previous chapter (1.1.1) we mentioned “sentences”, these sentences are expressed according to the **syntax** of the representation language, and needs to be clear, eg:

“X + Y = 8” is a well-formed sentence whereas “XY8+=” is not.

A logic must also define the **semantics** or meaning of the sentence, the semantics defines the **truth** of each sentence with respect to each **possible world**, in standard logics, evert sentence must be either true or false in each possible world, the is no “in-between”.

We use the term **model** in place of “possible world”, if a sentence A is true in model M, we say that M **satisfies** A, or M **is a model of** A.

logical reasoning involves the relation of logical **entailment** between sentences—the idea that a sentence follows logically from another sentence, the formal deﬁnition of entailment is this: A |= B if and only if, in every model in which A is true, B is also true.

The deﬁnition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**, one inference algorithm is called model checking, because it enumerates all possible models to check that A is true in all models in which KB is true.

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent A. It is easy to see that model checking, when it is applicable, is a sound procedure, the property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed.

The **syntax** of **propositional logic** defines the allowable sentences, **atomic sentences** consist of a single proposition symbol. Each such symbol stands for a proposition that can be true or false.

Complex sentences are constructed from simpler sentences, using parentheses and logical connectives, there are five connectives in common use;

Negation (¬), Conjunction (∧), Disjunction (∨), Implication (⇒) and Biconditional (⇔).

Focusing on propositional logic **semantics**, a model simply ﬁxes the truth value—true or false—for every proposition symbol, the semantics for propositional logic must specify how to compute the truth value of any sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the ﬁve connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the ﬁve connectives, for complex sentences;

• ¬ P is true iff P is false in m.

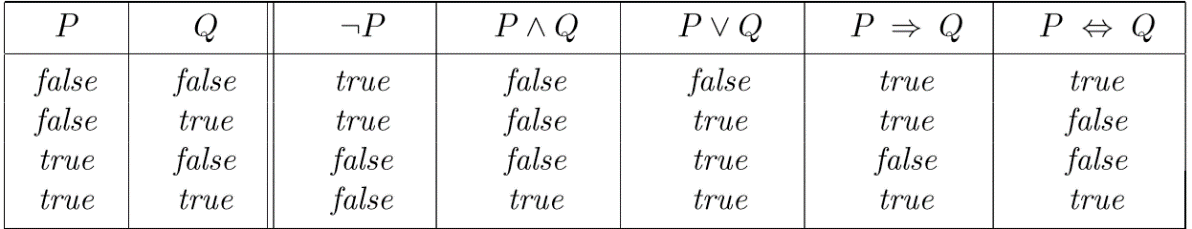
• P ∧ Q is true iff both P and Q are true in m.

• P ∨ Q is true iff either P or Q is true in m.

• P ⇒ Q is true unless P is true and Q is false in m.

• P ⇔ Q is true iff P and Q are both true or both false in m.

The rules can also be expressed with **truth tables** that specify the truth value of a complex sentence for each possible assignment of truth values to its components. E.g.



* + 1. **Propositional Theorem Proving**

So far, we have shown how to determine entailment by *model checking*: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models.

First, we will need some additional **concepts** related to entailment;

**Logical equivalence** – two sentences A and B are logically equivalent if they are true in the same set of models. We write this as A ≡ B.

**Validity** – a sentence is valid if it is true in *all* models, valid sentences are also known as **Tautologies** - they are necessarily true, what good are valid sentences? From our deﬁnition of entailment, we can derive the **deduction theorem**, which states that every valid implication sentence describes a legitimate inference. The ﬁnal concept we will need is **satisﬁability**. A sentence is satisﬁable if it is true in, or satisﬁed by, *some* model. Satisﬁability can be checked by enumerating the possible models until one is found that satisﬁes the sentence. The problem of determining the satisﬁability of sentences in propositional logic—the **SAT** problem—was the ﬁrst problem proved to be NP-complete.

Validity and satisﬁability are of course connected: A is valid iff ¬A is unsatisﬁable; contra positively, A is satisﬁable iff ¬A is not valid.

Another approach is called proof by **refutation** or proof by **contradiction**. One assumes a sentence B to be false and shows that this leads to a contradiction with known axioms A.

* + 1. **First Order Logics**

In this chapter, we examine ﬁrst-order logic, one which is sufﬁciently expressive to represent a good deal of our commonsense knowledge. It also either subsumes or forms the foundation of many other representation languages and has been studied intensively for many decades.

We will discuss the representation languages in general, the syntax and semantics of first-order logic. Models for ﬁrst-order logic are very interesting. First, they have objects in them! The domain of a model is the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object. Mathematically speaking, it doesn’t matter what these objects are—all that matters is how many there are in each particular model. Formally speaking, a relation is just the set of **tuples** of objects that are related. (A tuple is a collection of objects arranged in a ﬁxed order and is written with angle brackets surrounding the objects.). Strictly speaking, models in ﬁrst-order logic require **total functions**, that is, there must be a value for every input tuple.

We turn now to the syntax of ﬁrst-order logic. The basic syntactic elements of ﬁrst-order logic are the symbols that stand for objects, relations, and functions. The symbols, therefore, come in three kinds: **constant symbols**, which stand for objects; **predicate symbols**, which stand for relations; and **function symbols**, which stand for functions.

As in propositional logic, every model must provide the information required to determine if any given sentence is *true or false*. Thus, in addition to its objects, relations, and functions, each model includes an interpretation that speciﬁes exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

A **term** is a logical expression that refers to an object. Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object, the formal *semantics* of terms is straightforward. Consider a term f(t1,...,tn). The function symbol f refers to some function in the model (call it F); the argument terms refer to objects in the domain (call them d1,...,dn); and the term as a whole refers to the object that is the value of the function F applied to d1,...,dn. Now that we have both terms for referring to objects and predicate symbols for referring to relations, we can put them together to make atomic sentences that state facts. An **atomic sentence** (or **atom** for short) is formed from a predicate symbol optionally followed by parenthesized list of terms, such as - “Brother (Richard, John)”.

We can use **logical connectives** to construct more complex sentences, with the same syntax and semantics as in propositional calculus, **Universal quantiﬁcation** (∀), usually pronounced “For all …”, Thus, the sentence says, “For all x, if x is a king, then x is a person.” The symbol x is called a **variable,** a term with no variables is called a **ground term,** universal quantiﬁcation makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an **Existential quantiﬁcation** (∃), the two quantiﬁers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same as asserting there does not exist someone who likes them, and vice versa: “ ∀x ¬Likes(x,Parsnips) is equivalent to ¬∃x Likes(x,Parsnips) “.

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called **assertions**. Questions asked with ASK are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered afﬁrmatively.

* 1. **Planning**

We have deﬁned AI as the study of rational action, which means that planning—devising a plan of action to achieve one’s goals—is a critical part of AI.

Here we will cover a fully observable, deterministic, static environment with single agents.

planning researchers have settled on a factored representation— one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Deﬁnition Language, that allows us to express all PDDL actions with one action schema.

PDDL describes the four things we need to deﬁne a search problem: the **initial state**, the **actions** that are available in a state, the **result** of applying an action, and the **goal test**, each state is represented as a conjunction of ﬂuents that are ground, functionless atoms.

*Actions* are described by a set of action schemas that implicitly deﬁne the ACTIONS(s) and RESULT(s,a) functions needed to do a problem-solving search. **Classical planning** concentrates on problems where most actions leave most things unchanged. A set of ground (variable-free) actions can be represented by a single action schema. The schema is a lifted representation—it lifts the level of reasoning from propositional logic to a restricted subset of ﬁrst-order logic, the schema consists of the action name, a *list of all the variables used in the schema*, a **precondition** and an **effect**, PDDL was derived from the original **STRIPS** planning language (Fikes and Nilsson, 1971). which is slightly more restricted than PDDL: STRIPS preconditions and goals cannot contain negative literals. The precondition and effect of an action are each conjunctions of literals (positive or negated atomic sentences). The precondition deﬁnes the states in which the action can be executed, and the effect deﬁnes the result of executing the action. An action a can be executed in state s if s entails the precondition of a, we say that action a is applicable in state s if the preconditions are satisﬁed by s.

* + 1. **Algorithms for Planning**

We saw how the description of a planning problem deﬁnes a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state.

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms, **forward search** is prone to exploring irrelevant actions, while in **backward search**, we start at the goal and apply the actions backward until we ﬁnd a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). In general, backward search works only when we know how to regress from a state description to the predecessor state description. The PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it.

The ﬁnal issue is deciding which actions are candidates to regress over. In the forward direction we chose actions that were **applicable**—those actions that could be the next step in the plan. In backward search we want actions that are **relevant**—those actions that could be the last step in a plan leading up to the *current* goal state. For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action’s effects (either positive or negative) must unify with an element of the goal. What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal.

2. **Introducing the Problem**

The problem we will discuss and implement via PDDL and Planning.Domain, is the hospital problem, imagine a hospital with several rooms, nurses, doctors and treatments, which needs a schedule master that will schedule every ones day, whom to which room, what pill for who, operations and etc. The schedule master has a huge variety of sequences that can schedule every ones’ day without any conflicts (operation room is not available), this is a hard task for a planner but surely feasible.

2.1 **Defining the Problem**

The hospital schedule problem has one main goal – to complete a **day** after requiring every one **goals**. We will describe a “day” and “goals” in the next paragraph, first we need to understand what features our hospital has that can satisfy every ones needs. A hospital has **rooms**, which in our case, each **patient** has its own room and each type of **operation** has its own room, it also has **nurses** and **doctors**, in our situation, the nurses has several jobs they can fulfil – give a **meal** (breakfast, lunch and dinner) to a patient, give a pill (morning pill, afternoon pill and evening pill) to a patient, conveyance of the patient for a room to another, and check a patient. A doctor has one important role – operating a patient, which also requires specific tools for that operation. After explaining the hospital features, we now move to the patients, every patient has its own **treatments**, and **needs**, that needs to be satisfied, the main treatments that out hospital has is an operation and ultrasound, and the main need of a patient is eating. Completing a day in the hospital is when all the patients’ needs and treatments are satisfied. Notice that, many treatments and needs has **preconditions**, a patient can take a specific pill without eating a meal beforehand, can’t eat dinner before eating lunch, can’t get his operation unless he is in the operation room, several operations and pills needs a nurse checkup beforehand, now it is clear that the task “scheduling a hospital day” is very hard.

2.2 **The Domain**

A domain file in PDDL defines the “universal” aspects of a problem. Essentially, these are the aspects that do not change regardless of what specific situation we’re trying to solve. In PDDL this is mostly the object types, predicates and actions that can exist within the model, we define the model with (define (domain hospital) ), and within the “define” statement we can write the whole problem definition.

2.2.1 **The Requirements**

Requirements are similar to import/include statements in programming languages, however as PDDL is a kind of declarative language, it is a :requirement as a given planner is “required” to facilitate some aspect of the language, in our world, we will need only the STRIPS “library”, which Allows the usage of basic add and delete effects as specified in STRIPS, and adds the line

(:requirements :strips)

2.2.2 **The Predicates**

One thing that is important to understand is that, apart from the special predicate =, predicates in a domain definition have *no intrinsic meaning*. The :predicates part of a domain definition specify only what are the predicate names used in the domain, and their number of arguments (and argument types, if the domain uses typing). The "meaning" of a predicate, in the sense of for what combinations of arguments it can be true and its relationship to other predicates, is determined by the effects that actions in the domain can have on the predicate, and by what instances of the predicate are listed as true in the initial state of the problem definition. In the hospital problem the predicates are as followed:

(pill\_has\_allergies ?y) – do pill Y has ingredients that causes allergies ?

(pre\_pill\_checkup ?y) – do a patient who takes pill Y needs a checkup beforehand ?

(before\_breakfest\_pill ?y) – pill Y needs to be taken before breakfast ?

(after\_breakfest\_pill ?y) – pill Y needs to be taken after breakfast ?

(before\_lunch\_pill ?y) – pill Y needs to be taken before lunch ?

(after\_lunch\_pill ?y) – pill Y needs to be taken after lunch?

(before\_dinner\_pill ?y) – pill Y needs to be taken before dinner ?

(after\_dinner\_pill ?y) – pill Y needs to be taken after dinner?

(need\_to\_invite\_pill ?y) – the pill Y exists in the inventory or needs to be ordered ?

(need\_pill ?x ?y) – do patient X needs pill Y ?

(need\_usd ?x) – do patient X needs ultra-sound ?

(need\_op ?x) – do patient X needs to go through a surgery ?

(ate\_breakfest ?x) – do patient X ate breakfast ?

(ate\_lunch ?x) - do patient X ate lunch ?

(ate\_dinner ?x) - do patient X ate dinner ?

(patient\_has\_allergies ?x) – do patient X has ingredients he is allergic to ?

(all\_ate\_breakfest ?b) – do all patients ate breakfast ? (b is for Boolean)

(all\_ate\_lunch ?b) – do all patients ate breakfast ?

(all\_ate\_dinner ?b) – do all patients ate breakfast ?

(is\_all\_ate\_bool ?b) – do all patient ate the current meal ?

(is\_room ?r) – is R a room ?

(is\_patient ?x) – is X a patient ?

(is\_meal ?m) – is M a meal ?

(meals\_prepared ?b) – do all meals are prepared to be served ?

(is\_usd\_room ?r) – is R a ultra-sound room ?

(is\_op\_room ?r) – is R a surgery room ?

(is\_doctor ?d) – is D a doctor ?

(is\_nurse ?n) – is N a nurse ?

(is\_nurse\_room ?r) – is R a nurse-room ?

(is\_doctor\_room ?r) – is R a doctor-room ?

(doctor\_need\_tools ?d) – do doctor D needs tools ? (to perform a surgery)

(nurse\_need\_tools ?n) – do nurse N needs tools ?

(has\_tools ?dn) – do DN has tools ? (DN can be doctor or a nurse)

(doctor\_at\_room ?d ?x) – is doctor D in room X ?

(nurse\_at\_room ?n ?x) – is nurse N in room X ?

2.2.3 **The Actions**

All parts of an action definition except the name are, according to the PDDL specification, optional (although, of course, an action without effects is pretty useless).

(:action take\_pill

:parameters (?x ?y ?n)

:precondition(or

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (patient\_has\_allergies ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_breakfest ?x)) (not(ate\_lunch ?x)) (not(ate\_dinner ?x)) (before\_breakfest\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (patient\_has\_allergies ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_lunch ?x)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (after\_breakfest\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (patient\_has\_allergies ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_lunch ?x)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (before\_lunch\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (patient\_has\_allergies ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (ate\_lunch ?x) (after\_lunch\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (patient\_has\_allergies ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (ate\_lunch ?x) (before\_dinner\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (patient\_has\_allergies ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (ate\_breakfest ?x) (ate\_lunch ?x) (ate\_dinner ?x) (after\_dinner\_pill ?y))

**This action is very long, and subtracted into 2 parts, the upper part is taking a pill with allergies, and the bottom part is taking a pill that has no allergies**

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_breakfest ?x)) (not(ate\_lunch ?x)) (not(ate\_dinner ?x)) (before\_breakfest\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_lunch ?x)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (after\_breakfest\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_lunch ?x)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (before\_lunch\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (ate\_lunch ?x) (after\_lunch\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (not(ate\_dinner ?x)) (ate\_breakfest ?x) (ate\_lunch ?x) (before\_dinner\_pill ?y))

(and (is\_patient ?x)(is\_nurse ?n) (nurse\_at\_room ?n ?x) (not(pre\_pill\_checkup ?y)) (not(need\_to\_invite\_pill ?y)) (not(pill\_has\_allergies ?y)) (ate\_breakfest ?x) (ate\_lunch ?x) (ate\_dinner ?x) (after\_dinner\_pill ?y)))

:effect(and

(not(need\_pill ?x ?y))))

**This action, takes care the whole procedure of delivering, and taking pills, in our hospital, patients need pills, in order to take a pill, there are many preconditions that’s need to be satisfied, there must be a patient who needs a pill, and if that patient has allergies, he needs a special pill that needs to be ordered beforehand (ordering a pill is an action as well), do that patient needs to be checked by a nurse before taking the pill? Does the pill requires that the patient ate a meal before taking the pill? And of curse a nurse that will deliver him the pill.**

(:action eat\_breakfest

:parameters (?x ?m ?n)

:precondition(and

(is\_patient ?x)(is\_nurse ?n)(nurse\_at\_room ?n ?x)(is\_meal ?m)(meals\_prepared ?b)

(not(ate\_breakfest ?x))

:effect(and

(ate\_breakfest ?x)))

:action eat\_lunch

:parameters (?x ?m ?b ?n)

:precondition(and

(is\_patient ?x)(is\_nurse ?n)(nurse\_at\_room ?n ?x)(is\_all\_ate\_bool ?b)(is\_meal ?m)

(meals\_prepared ?b)(all\_ate\_breakfest ?b)(not(ate\_lunch ?x))

:effect(and

(ate\_lunch ?x)))

(:action eat\_dinner

:parameters (?x ?m ?b ?n)

:precondition(and

(is\_patient ?x)(is\_nurse ?n)(nurse\_at\_room ?n ?x)(is\_all\_ate\_bool ?b)(is\_meal ?m)

(meals\_prepared ?b)(all\_ate\_lunch ?b)(not(ate\_dinner ?x))

:effect(and

(ate\_dinner ?x)))

**These actions, takes care of the patient meals, every patient needs to finish his day after he got served with dinner (of curse dinner cannot be server unless he got already breakfast and then lunch), for a patient to get served with a meal, the meal needs to be prepared and a nurse needs to deliver the meal to his room, notice that no one can eat a current meal unless everyone took the last meal.**

:action check\_before\_taking\_pill

:parameters (?x ?y ?n)

:precondition(and

(need\_pill ?x ?y)(pre\_pill\_checkup ?y)(is\_nurse ?n)(nurse\_at\_room ?n ?x)

:effect(and

(not(pre\_pill\_checkup ?y))))

**This action deals with the checkup that the nurse needs to preform before giving a pill to a patient who classified in “delicate situation”, and needs to be checked before taking a pill.**

:action invite\_pill

:parameters (?y ?r ?n)

:precondition(and

(is\_nurse ?n)(is\_nurse\_room ?r)(nurse\_at\_room ?n ?r)(need\_to\_invite\_pill ?y))

:effect(and

(not(need\_to\_invite\_pill ?y))))

**This action, invites pill that is missing from the inventory, in order to order a pill, the nurse has to be in the nurse room.**

:action get\_no\_allergie\_pill

:parameters(?y ?r ?n)

:precondition(and

(is\_nurse ?n)(is\_nurse\_room ?r)(nurse\_at\_room ?n ?r)(pill\_has\_allergies ?y))

:effect(and

(not(pill\_has\_allergies ?y))))

**Whenever a patient with allergies needs a specific pill that contain ingredients his allergic to, the nurse needs to go to the nurse room and fetch a special pill that the patient could take.**

:action nurse\_go\_to\_room

:parameters (?n ?x ?r)

:precondition(and

(is\_room ?r)(is\_nurse ?n)(not(nurse\_at\_room ?n ?x))(nurse\_at\_room ?n ?r))

:effect(and

(nurse\_at\_room ?n ?x)(not(nurse\_at\_room ?n ?r)))

:action doctor\_go\_to\_room

:parameters (?d ?x ?r)

:precondition (and

(is\_room ?r)(is\_doctor ?d)(not(doctor\_at\_room ?d ?x))(doctor\_at\_room ?d ?r)

:effect (and

(doctor\_at\_room ?d ?x)(not(doctor\_at\_room ?d ?r)))

**This two actions, takes care of the doctor and nurse transportation (between rooms), notice that a doctor can’t go to a room that he is already in.**

:action do\_op

:parameters (?x ?r ?d)

:precondition (and

(not(ate\_breakfest ?x))(need\_op ?x)(is\_doctor ?d)(is\_op\_room ?r)

(doctor\_at\_room ?d ?r)(has\_tools ?d)

:effect (and

(not(need\_op ?x))(not(has\_tools ?d))))

(:action do\_usd

:parameters (?x ?r ?n)

:precondition (and

(not(ate\_breakfest ?x))(is\_nurse ?n)(need\_usd ?x)(is\_usd\_room ?r)

(nurse\_at\_room ?n ?r)(has\_tools ?n))

:effect (and

(not(need\_usd ?x))(not(has\_tools ?n))))

**These two actions are take care of the special operations a patient needs, in order to do those special operations, the patient must be fasting, therefore, the special operation must be the at the beginning.**

(:action get\_tools

:parameters (?nd ?r)

:precondition (and

(or(and(nurse\_at\_room ?nd ?r)(is\_nurse\_room ?r)(is\_nurse ?nd))

(and(is\_doctor ?nd)(is\_doctor\_room ?r)(doctor\_at\_room ?nd ?r)))(not(has\_tools ?nd)))

:effect (and

(has\_tools ?nd)))

There are operations that needs special tools, in order for a doctor or a nurse to perform an such operations, they have to get the tools.

(:action approve\_all\_ate\_breakfest

:parameters (?x1 ?x2 ?m ?b)

:precondition (and

(is\_all\_ate\_bool ?b)(is\_meal ?m)(meals\_prepared ?b)(not(= ?x1 ?x2))

(not(all\_ate\_breakfest ?b))(ate\_breakfest ?x1)(ate\_breakfest ?x2))

:effect (and

(all\_ate\_breakfest ?b)(not(meals\_prepared ?b))))

(:action approve\_all\_ate\_lunch

:parameters (?x1 ?x2 ?m ?b)

:precondition (and

(is\_all\_ate\_bool ?b)(is\_meal ?m)(meals\_prepared ?b)(not(= ?x1 ?x2))

(not(all\_ate\_lunch ?b))(ate\_lunch ?x1)(ate\_lunch ?x2))

:effect (and

(all\_ate\_lunch ?b)(not(meals\_prepared ?b))))

(:action approve\_all\_ate\_dinner

:parameters (?x1 ?x2 ?m ?b)

:precondition (and

(is\_all\_ate\_bool ?b)(is\_meal ?m)(meals\_prepared ?b)(not(= ?x1 ?x2))

(not(all\_ate\_dinner ?b))(ate\_dinner ?x1)(ate\_dinner ?x2))

:effect (and

(not(meals\_prepared ?b))(all\_ate\_dinner ?b)))

**These three actions, made to set the order of the day at the hospital, before these actions were made, the planner could give a patient all the meals he needed at the same time, so in order to set the flow, we needed to stop the planner from giving a meal to a patient if all the other patients didn’t had the last meal.**

(:action prepare\_meals

:parameters (?n ?r ?m)

:precondition (and

(is\_meal ?m)(is\_nurse ?n)(is\_nurse\_room ?r)(nurse\_at\_room ?n ?r)(not(meals\_prepared ?b)))

:effect (and

(meals\_prepared ?b)))

**These action deal with the meal preparation, in order for the meal the be ready, a nurse needs to go to the nurse room and make all the meals.**

2.3 **The Problem**

As explained before, our problem is the “hospital scheduling problem”, in order to get to the maximum power of our planner (planner domain), we have tried several implementation of the problem e.g. different amount of doctors and nurses, treatments and pills each patient needs, number of patient and so on. In this chapter we will demonstrate a simple implementation of the problem, in the next chapter we will review more complicated implementation that “suffocates” the planner.

2.3.1 **Objects**

In our simple model, we have 1 nurse, 1 doctor, 2 patients, 3 pill which needed to be taken by the patients, an ultrasound room, an operation room, a nurse room, a doctor room, a chill room (which the nurses need to be at the end of the day), and two “Booleans” – meal and allatebool.

nurse1 – the nurse.

per1 per2 – the patients.

pil1 pil2 pil3 – the pills.

doctor1 – the doctos.

usdroom oproom chillroom doctorroom nurseroom1 – all the rooms.

meal allatebool – the boolean variables.

2.3.2 **Initialization**

The initialization part in the problem section, is the condition of the hospital at the start of the day, in this part, we will have to declare all the rooms, the doctors and the nurses, pills, meals and so on.

(is\_nurse nurse1) (nurse\_at\_room nurse1 chillroom) **– initializing the nurse and where she is at the start of the day**.

(is\_doctor doctor1) (doctor\_at\_room doctor1 chillroom) – **initializing the doctor and where he is at the start of the day.**

(is\_patient per1)(is\_patient per2) – **initializing the patients.**

(is\_room nurseroom1) (is\_room doctorroom)(is\_room usdroom) (is\_room oproom)

(is\_room chillroom) (is\_room per1) (is\_room per2) – **initializing all the rooms as rooms in the hospital.**

(is\_meal meal)

(is\_all\_ate\_bool allatebool) – **initializing the boolean variables.**

(is\_usd\_room usdroom)

(is\_op\_room oproom)

(is\_nurse\_room nurseroom1)

(is\_doctor\_room doctorroom) – **initializing each room with his type.**

(patient\_has\_allergies per1) – **initializing which patient has allergies.**

(after\_lunch\_pill pil2)

(need\_to\_invite\_pill pil2)

(pill\_has\_allergies pil2)

(before\_breakfest\_pill pil1)

(pill\_has\_allergies pil1)

(pre\_pill\_checkup pil1)

(after\_dinner\_pill pil3)

(need\_to\_invite\_pill pil3) – **initializing each pill with the time it should be taken, if it has ingredients that causes allergies (to patient that has allergies), and whether the patient needs a nurse checkup before taking the pill, and whether the pill is in the inventory or need to be orderd.**

(need\_op per1)

(need\_usd per2) – **initializing which patient needs what operation**

(need\_pill per1 pil1)

(need\_pill per1 pil2)

(need\_pill per2 pil2)

(need\_pill per2 pil3) - **initialing each patient with the pills he will need.**

2.3.3 **The goals**

In this section, we will list all the goals that’s need to be done in order to finish our run.

(not(need\_pill per1 pil1)) (not(need\_pill per1 pil2))

(not(need\_pill per2 pil1)) (not(need\_pill per2 pil2)) (not(need\_pill per2 pil3)) – **All the patients should take all the pills they needed.**

(not(need\_op per1)) (not(need\_usd per2)) – **the operations that the patients needed should be done.**

(doctor\_at\_room doctor1 chillroom)

(nurse\_at\_room nurse1 chillroom) - **saying all the staff should be at the chill room at the end of the day.**

Notice that, there are many actions that needs to be done that isn’t written here, in order to be able to finish the task properly, such as eating meals, transporting the staff, and many more.

3. **Planner Implementation**

3.1 **The Simple Case Output.**

The simple case is the one we reviewed in the last chapter, we can say that the planner had no difficulty to come up with the solution.

the planner solution information – match tree built with 266 nodes.

nodes generated : 28759, nodes expanded : 817, total time : 1.32.

We can see that the planner came up with a plan very fast, we want to make it harder for the planner to come up with the plan (full planner output is attached).

3.2 **More Cases and Analysis.**

3.2.1 **Case 1**

In this case, we have everything the same as the last problem, but with 2 doctors instead of one. We can see that whenever we are adding more “space of solution” to the planner, it takes more time to come up with the right plan, that’s because the planner can’t remove the same amount of “branches” in this search tree as before, he got a lot more “open space”

Planners output – match tree built 387 nodes.

Nodes generated : 74687, nodes expanded : 1427, total time : 5.42

3.2.2 **Case 2**

Same as first case but this time with 3 doctors, in this case, the planner couldn’t find the solution, that’s because planner.domain has a time limit of 10 seconds, that’s mean it took more then 10 seconds.

3.2.3 **Case 3**

In order to go down under 10 seconds in the last case, we needed to give up on some constraints, in this case, we gave up on a “small” constraint and the planner managed to come up with a solution, each staff at the end of the day had to go to the chill room, we gave the 3rd doctor a break and he could finish his shift wherever he wanted, this move gave us the insight the old case wasn’t much higher then 10 seconds. Here is the information on the solution – nodes generated : 2790473, nodes expanded : 4171, total time : 9.056

3.3 **Conclusion**

After many cases we have tried (more then 3), we understood that the more “open space” we are giving to the planner, the harder it gets for him to get a solution, furthermore, adding more operations and pills didn’t really make it hard for the planner, adding more constraints also made it easier to the planner to find the solution (removing a lot of branches within the search). In order to write the perfect domain and problem, it is important to understand the “world” of the problem, what are the variables and situations, conducting experiments, write short actions, and work with variable type (which we didn’t do). Understanding the world of NP problems, and clustering them into group of solutions helped us a lot in order to classify our problem and come up with a way of solving it.