DEPARTMENT OF COMPUTER SCIENCE AND ENGG.

INTELLIGENT SYSTEM

SEM-6TH

SECTION-C (NOTES)

Reasoning system

In information technology a **reasoning system** is a software system that generates conclusions from available knowledge using logical techniques such as deduction and induction. Reasoning systems play an important role in the implementation of artificial intelligence and knowledge-based systems.

By the everyday usage definition of the phrase, all computer systems are reasoning systems in that they all automate some type of logic or decision. In typical use in the Information Technology field however, the phrase is usually reserved for systems that perform more complex kinds of reasoning. For example, not for systems that do fairly straightforward types of reasoning such as calculating a sales tax or customer discount but making logical inferences about a medical diagnosis or mathematical theorem. Reasoning systems come in two modes: interactive and batch processing. Interactive systems interface with the user to ask clarifying questions or otherwise allow the user to guide the reasoning process. Batch systems take in all the available information at once and generate the best answer possible without user feedback or guidance

Reasoning systems have a wide field of application that includes scheduling, business rule processing, problem solving, complex event processing, intrusion detection, predictive analytics, robotics, computer vision, and natural language processing.

The term reasoning system can be used to apply to just about any kind of sophisticated decision system as illustrated by the specific areas described below. However, the most common use of the term reasoning system implies the computer representation of logic. Various implementations demonstrate significant variation in terms of systems of logic and formality. Most reasoning systems implement variations of propositional and symbolic (predicate) logic. These variations may be mathematically precise representations of formal logic systems (e.g., FOL), or extended and hybrid versions of those systems. Reasoning systems may explicitly implement additional logic types (e.g., modal, deontic, temporal logics). However, many reasoning systems implement imprecise and semi-formal approximations to recognised logic systems. These systems typically support a variety of procedural and semi-declarative techniques in order to model different reasoning strategies. They emphasise pragmatism over formality and may depend on custom extensions and attachments in order to solve real-world problems.

SITAM REWARI

Many reasoning systems employ deductive reasoning to draw inferences from available knowledge. These inference engines support forward reasoning or backward reasoning to infer conclusions via modus ponens. The recursive reasoning methods they employ are termed 'forward chaining' and 'backward chaining', respectively. Although reasoning systems widely support deductive inference, some systems employ abductive, inductive, defeasible and other types of reasoning. Heuristics may also be employed to determine acceptable solutions to intractable problems.

Reasoning systems may employ the closed world assumption (CWA) or open world assumption (OWA). The OWA is often associated with ontological knowledge representation and the Semantic Web. Different systems exhibit a variety of approaches to negation. As well as logical or bitwise complement, systems may support existential forms of strong and weak negation including negation-as-failure and 'inflationary' negation (negation of non-ground atoms). Different reasoning systems may support monotonic or non-monotonic reasoning, stratification and other logical techniques.

Reasoning under uncertainty

Many reasoning systems provide capabilities for reasoning under uncertainty. This is important when building situated reasoning agents which must deal with uncertain representations of the world. There are several common approaches to handling uncertainty. These include the use of certainty factors, probabilistic methods such as Bayesian inference or Dempster–Shafer theory, multi-valued ('fuzzy') logic and various connectionist approaches.

STATISTICAL REASONING

There are several techniques that can be used to augment knowledge representation techniques with statistical measures that describe levels of evidence and belief . an important goal for many problem solving systems is to collect evidence as the systems goes along and to modify its behavior , we need a statistical theory of evidence. Bayesian statistics is such a theory which stresses the conditional probability as fundamental notion.

FUZZY LOGIC

In fuzzy logic, we consider what happens if we make fundamental changes to our idea of set membership and corresponding changes to our definitions of logical operations. While traditional set-theory defines set membership as a Boolean predicate, fuzzy set theory allows us to represent set membership as a possibility distribution such as tall-very for the set of tall people and the set of very tall people. This contrasts with the standard Boolean definition for tall people where one is either tall or not and there must be a specific height that defines the boundary. The same is true for very tall. In fuzzy logic, one's tallness increases with one's height until the value 1 is reached. So it is a distribution. Once set membership has been redefined in this way, it is possible to

SITM REWARI

define a reasoning system based on techniques for combining distributions. Such reasoners have been applied in control systems for devices as diverse as trains and washing machines.

Fuzzy Logic Systems (FLS) produce acceptable but definite output in response to incomplete, ambiguous, distorted, or inaccurate (fuzzy) input.

What is Fuzzy Logic

Fuzzy Logic (FL) is a method of reasoning that resembles human reasoning. The approach of FL imitates the way of decision making in humans that involves all intermediate possibilities between digital values YES and NO.

The conventional logic block that a computer can understand takes precise input and produces a definite output as TRUE or FALSE, which is equivalent to human's YES or NO.

The inventor of fuzzy logic, Lotfi Zadeh, observed that unlike computers, the human decision making includes a range of possibilities between YES and NO, such as −

The fuzzy logic works on the levels of possibilities of input to achieve the definite output.

Implementation

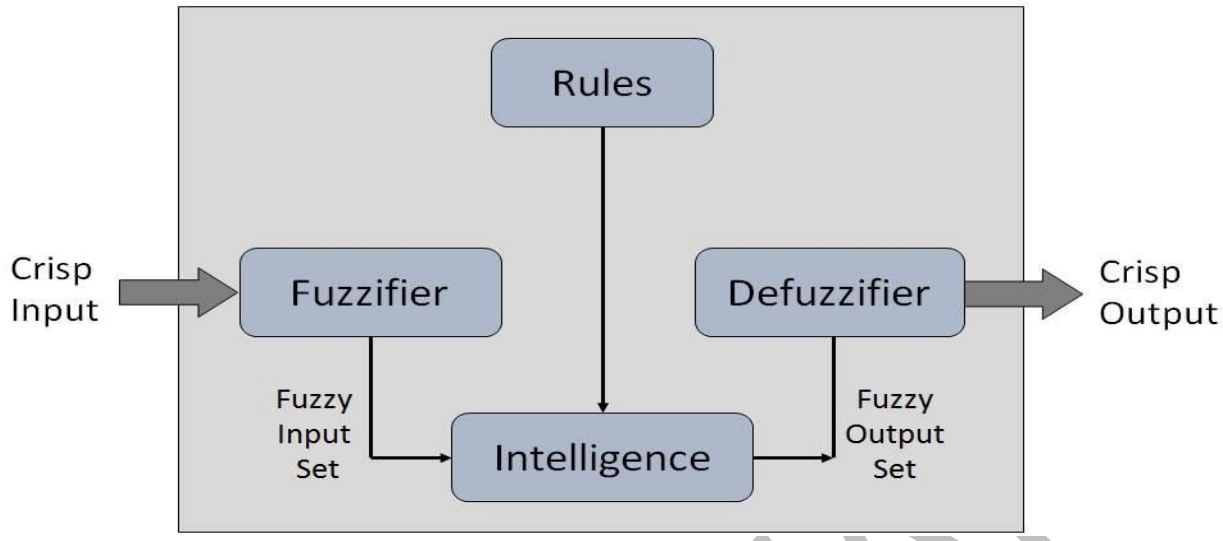
- It can be implemented in systems with various sizes and capabilities ranging from small micro-controllers to large, networked, workstation-based control systems.
- It can be implemented in hardware, software, or a combination of both.

Why Fuzzy Logic

Fuzzy logic is useful for commercial and practical purposes.

- It can control machines and consumer products.
- It may not give accurate reasoning, but acceptable reasoning.
- Fuzzy logic helps to deal with the uncertainty in engineering.
- **Knowledge Base** − It stores IF-THEN rules provided by experts.
- **Inference Engine** − It simulates the human reasoning process by making fuzzy inference on the inputs and IF-THEN rules.
- **Defuzzification Module** − It transforms the fuzzy set obtained by the inference engine into a crisp value.

SITM REWARI



The **membership functions work on** fuzzy sets of variables.

Membership Function

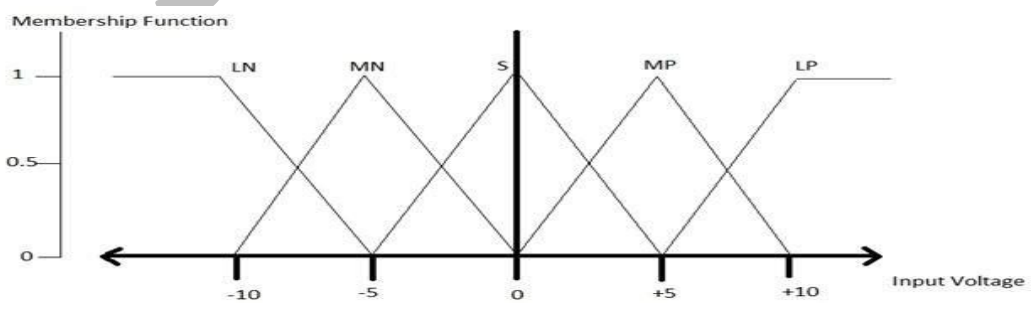
Membership functions allow you to quantify linguistic term and represent a fuzzy set graphically. A **membership function** for a fuzzy *set A* on the universe of discourse X is defined as $\mu A: X \rightarrow [0,1]$.

Here, each element of *X* is mapped to a value between 0 and 1. It is called **membership value** or **degree of membership**. It quantifies the degree of membership of the element in *X* to the fuzzy set *A*.

- x axis represents the universe of discourse.
- y axis represents the degrees of membership in the [0, 1] interval.

There can be multiple membership functions applicable to fuzzify a numerical value. Simple membership functions are used as use of complex functions does not add more precision in the output.

All membership functions for **LP, MP, S, MN,** and **LN** are shown as below −



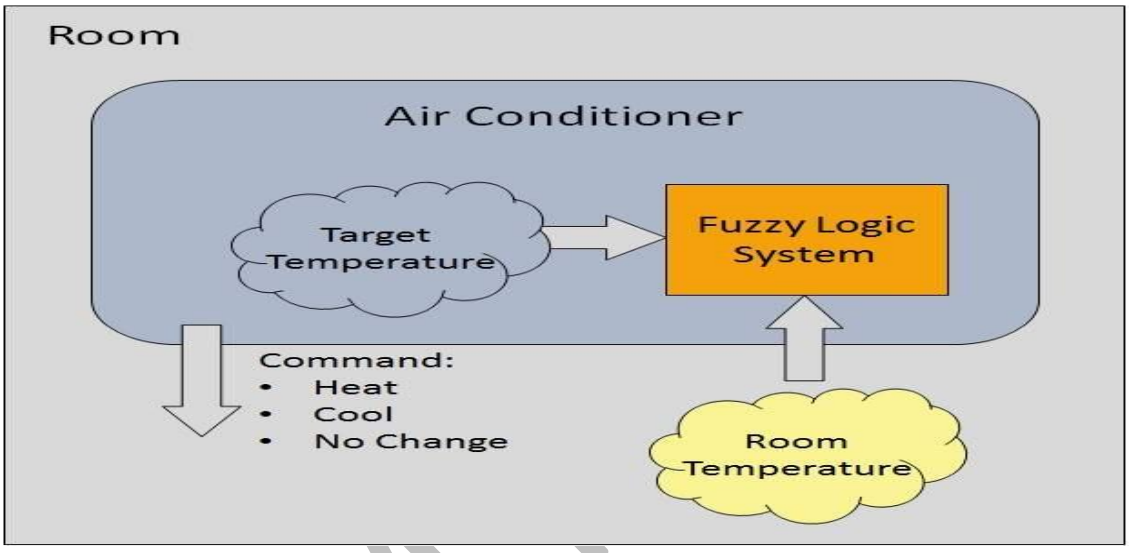
SIM REWA

The triangular membership function shapes are most common among various other membership function shapes such as trapezoidal, singleton, and Gaussian.

Here, the input to 5-level fuzzifier varies from -10 volts to +10 volts. Hence the corresponding output also changes.

Example of a Fuzzy Logic System

Let us consider an air conditioning system with 5-level fuzzy logic system. This system adjusts the temperature of air conditioner by comparing the room temperature and the target temperature value.



Algorithm

- Define linguistic variables and terms.
- Construct membership functions for them.
- Construct knowledge base of rules.
- Convert crisp data into fuzzy data sets using membership functions. (fuzzification)
- Evaluate rules in the rule base. (Inference Engine)
- Combine results from each rule. (Inference Engine)
- Convert output data into non-fuzzy values. (defuzzification)

Spatial–temporal reasoning is an area of artificial intelligence which draws from the fields of computer science, cognitive science, and cognitive psychology. The theoretic goal—on the cognitive side—involves representing and reasoning spatial-temporal knowledge in mind. The applied goal—on the computing side—involves developing high-level control systems of robots for navigating and understanding time and space.

Non-monotonic Reasoning

The definite clause logic is **monotonic** in the sense that anything that could be concluded before a clause is added can still be concluded after it is added; adding knowledge does not reduce the set of propositions that can be derived.

A logic is **non-monotonic** if some conclusions can be invalidated by adding more knowledge. The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. A **default** is a rule that can be used unless it overridden by an exception.

For example, to say that *b* is normally true if *c* is true, a knowledge base designer can write a rule of the form

$b \leftarrow c \wedge \sim ab_a.$

where ab_a is an atom that means abnormal with respect to some aspect *a*. Given *c*, the agent can infer *b* unless it is told ab_a. Adding ab_a to the knowledge base can prevent the conclusion of *b*. Rules that imply ab_a can be used to prevent the default under the conditions of the body of the rule.

Traditional systems based on predicate logic are monotonic . Here number of statements known to be true increases with time. New statements are added and new theorems are proved, but the previously known statements never become invalid.

In monotonic systems there is no need to check for in consistencies between new statements and the old knowledge. When a proof is made , the basis of the proof need not be remembered, since the old statements never disappear. But monotonic systems are not good in real problem domains where the information is incomplete, situations change and new assumptions are generated while solving new problems.

Non monotonic reasoning is based on default reasoning or “most probabilistic choice”. S is assumed to be true as long as there is no evidence to the contrary. For example when we visit a friend’s home, we buy biscuits for the children . because we believe that most children like biscuits. In this case we do not have information to the contrary. A computational description of default reasoning must relate the lack of information on X to conclude on Y.

Default reasoning (or most probabilistic choice) is defined as follows:

SJM REWARI

Definition 1 : If X is not known, then conclude Y.

Definition 2 : If X can not be proved, then conclude Y.

Definition 3: If X can not be proved in some allocated amount of time then conclude Y.

It is to be noted that the above reasoning process lies outside the realm of logic. It conclude on Y if X can not be proved, but never bothers to find whether X can be proved or not. Hence the default reasoning systems can not be characterized formally. Even if one succeeds in gaining complete information at the moment, the validity of the information may not be for ever, since it is a changing world. What appears to be true now, may be so at a later time (in a non monotonic system).

One way to solve the problem of a changing world to delete statements when they are no longer accurate, and replace them by more accurate statements. This leads to a non monotonic system in which statements can be deleted as well as added to the knowledge base. When a statement is deleted, other related statements may also have to be deleted. Non monotonic reasoning systems may be necessary due to any of the following reasons.

- The presence of incomplete inforamtio0n requires default reasoning.

- A changing world must be described by a changing database.

- Generating a complete solution to a problem may require temporary
  Assumptions about partial solutions.

Non monotonic system is harder to deal with than monotonic systems. This is because when a statement is deleted as “no more valid”, other related statements have to be backtracked and they should be either deleted or new proofs have to be found for them. This is called dependency directed backtracking (DDB). In order to propagate the current changes into the database, the statements on which a particular proof depends, should also be stored along with the proof. Thus non – monotonic systems require more storage space as well as more processing time than monotonic systems.

PLANNING

Planning is a key ability for intelligent systems, increasing their autonomy and flexibility through the construction of sequences of actions to achieve their goals. It has been an area of research in artificial intelligence for over three decades. Planning techniques have been applied in a variety of tasks including robotics, process planning, web-based information gathering, and autonomous agents and spacecraft mission control.

Planning involves the representation of actions and world models, reasoning about the effects of actions, and techniques for efficiently searching the space of possible plans. This course will focus on the basic foundations and techniques in planning and survey a variety of planning

SIM REWARI

systems and approaches. The class will be run as a lecture course with hands-on experience with state-of-the-art planning systems. Topics covered in the course will include: action and plan representation, reactive systems, hierarchical and abstraction planning, case-based planning, machine learning in planning, multi-agent planning, interacting with the environment, planning under uncertainty, and recent applications such as web service composition and workflow construction on the computational Grid.

The *planning* problem in Artificial Intelligence is about the decision making performed by intelligent creatures like robots, humans, or computer programs when trying to achieve some goal. It involves choosing a sequence of actions that will (with a high likelihood) transform the state of the world, step by step, so that it will satisfy the goal. The world is typically viewed to consist of atomic *facts* (state variables), and actions make some facts true and some facts false. In the following we discuss a number of ways of formalizing planning, and show how the planning problem can be solved automatically.

The most basic planning problem is one instance of the general s-t reachability problem for succinctly represented transition graphs, which has other important applications in Computer Aided Verification (reachability analysis, model-checking), Intelligent Control, discrete event-systems diagnosis, and so on. All of the methods described below are equally applicable to all of these other problems as well, and many of these methods were initially developed and applied in the context of these other problems.

The methods discussed in this tutorial have strengths in different types of problems.

- Symbolic methods based on BDDs excel in problems with a relatively small number of state variables (up to one or two hundred), with a complex but regular state space.

- Explicit state-space search is generally limited to small state spaces, but the AI planning community has been successfully applying explicit state-space search also to very large state-spaces, when their structure is simple enough to allow useful heuristic distance estimates, and when there are plenty of plans to choose from.

- Methods based on logic and constraints (SAT, constraint programming) are strong on problems with relatively high numbers of state variables and not too long plans, especially when constraints about the structure of the solution plans and the reachable state-space are available.

SIM REWARI

Models of state transition systems

Most works on planning use a *state-transition system* model (we often just write *transition system*), in which the world/system consists of a (finite or infinite) number of states, and actions/transitions change the current state to a next state. (Exceptions to this are planning with

Hierarchical Task Networks (HTN), and planning problems closer to Scheduling, in which the world states are not represented explicitly.)

There are several possible representations for state-transition systems. A state-transition system can be represented as an arc-labeled directed multi-graph. Each *node* of the graph is a *state*, and the arcs represent actions. An "action" is usually something that can be taken in several states, with the same or different effects. All the arcs corresponding to one action are *labeled* with the name of the action. There may be more than one action that moves us from state A to state B, and this means that there are more than one arc from A to B (this is why we said that the graph is a multi-graph.)

When states are represented as valuations of state variables, an action can be represented as a *procedure* or a *program* for changing the values of the state variables by making value assignments to them. Below we explain a couple of possibilities how these procedures can be defined.

Each of these action representations describes a *binary relation* on the set of all states: what are the possible pairs *(s,s')* of current state *s* and its successor state *s'*.

STRIPS

The simplest language used for formalizing actions is the STRIPS language. In STRIPS, the state variables have the domain {0,1} (equivalently {FALSE, TRUE}), and an action consists of three sets of state variables, the PRECONDITION, the ADD=$\{a_1,a_2,...,a_n\}$ list, and the DELETE=$\{d_1,d_2,...,d_m\}$ list (it is assumed that ADD and DELETE don't intersect.)

An action is possible in a state if all the variables in PRECONDITION have the value 1. Taking the action corresponds to executing the following program, consisting of assignment statements only:
$a_1 := 1$
$a_2 := 1$
...
$a_n := 1$
$d_1 := 0$
$d_2 := 0$
...
$d_m := 0$.

All the assignments are instantaneous and take place simultaneously. In STRIPS planning, a *goal* is usually expressed as a set of state variables. A state is a *goal state* if all of the goals have the value 1 in it.

SITM REWARI

PLANNING IN SITUATION CALCULUS

The idea behind **situation calculus** is that (reachable) states are definable in terms of the actions required to reach them. These reachable states are called situations. What is true in a situation can be defined in terms of relations with the situation as an argument. Situation calculus can be seen as a relational version of the feature-based representation of actions.

Here we only consider single agents, a fully observable environment, and deterministic actions.

Situation calculus is defined in terms of situations. A **situation** is either

- *init*, the initial situation, or
- *do(A,S)*, the situation resulting from doing action *A* in situation *S*, if it is possible to do action *A* in situation *S*.

A situation can be associated with a state. There are two main differences between situations and states:

- Multiple situations may refer to the same state if multiple sequences of actions lead to the same state. That is, equality between situations is not the same as equality between states.
- Not all states have corresponding situations. A state is **reachable** if a sequence of actions exists that can reach that state from the initial state. States that are not reachable do not have a corresponding situation.

Some *do(A,S)* terms do not correspond to any state. However, sometimes an agent must reason about such a (potential) situation without knowing if *A* is possible in state *S*, or if *S* is possible.

A **static** relation is a relation for which the truth value does not depend on the situation; that is, its truth value is unchanging through time. A **dynamic** relation is a relation for which the truth value depends on the situation. To represent what is true in a situation, predicate symbols denoting dynamic relations have a situation argument so that the truth can depend on the situation. A predicate symbol with a situation argument is called a **fluent**.

A dynamic relation is axiomatized by specifying the situations in which it is true. Typically, this is done inductively in terms of the structure of situations.

- Axioms with *init* as the situation parameter are used to specify what is true in the initial situation.
- A **primitive** relation is defined by specifying when it is true in situations of the form *do(A,S)* in terms of what is true in situation *S*. That is, primitive relations are defined in terms of what is true at the previous situation.

SIMREWARI

- A **derived** relation is defined using clauses with a variable in the situation argument. The truth of a derived relation in a situation depends on what else is true in the same situation.

- Static relations are defined without reference to the situation

Partial-order planning

Partial-order planning is an approach to automated planning that leaves decisions about the ordering of actions as open as possible. It contrasts with **total-order planning**, which produces an exact ordering of actions. Given a problem in which some sequence of actions is required in order to achieve a goal, a **partial-order plan** specifies all actions that need to be taken, but specifies an ordering of the actions only where necessary.

Partial-order plan

A **partial-order plan** or **partial plan** is a plan which specifies all actions that need to be taken, but does not specify an exact order for the actions when the order does not matter. It is the result of a partial-order planner. A partial-order plan consists of four components:

- A set of **actions** (also known as **operators**).

- A **partial order** for the actions. It specifies the conditions about the order of some actions.

- A set of **causal links**. It specifies which actions meet which preconditions of other actions. Alternatively, a set of **bindings** between the variables in actions.

- A set of **open preconditions**. It specifies which preconditions are not fulfilled by any action in the partial-order plan.

In order to keep the possible orders of the actions as open as possible, the set of order conditions and causal links must be as small as possible.

A plan is a solution if the set of open preconditions is empty.

A **linearization** of a partial order plan is a total order plan derived from the particular partial order plan; in other words, both order plans consist of the same actions, with the order in the linearization being a linear extension of the partial order in the original partial order pla

The forward and regression planners enforce a total ordering on actions at all stages of the planning process. The CSP planner commits to the particular time that the action will be carried out. This means that those planners have to commit to an ordering of actions that cannot occur concurrently when adding them to a partial plan, even if there is no particular reason to put one action before another.

SIM REWARI

The idea of a **partial-order planner** is to have a partial ordering between actions and only commit to an ordering between actions when forced. This is sometimes also called a **non-linear planner**, which is a misnomer because such planners often produce a linear plan.

A partial ordering is a less-than relation that is transitive and asymmetric. A **partial-order plan** is a set of actions together with a partial ordering, representing a "before" relation on actions, such that any total ordering of the actions, consistent with the partial ordering, will solve the goal from the initial state. Write $act_0 < act_1$ if action act_0 is before action act_1 in the partial order. This means that action act_0 must occur before action act_1.

Partial-order planner

A **partial-order planner** is an algorithm or program which will construct a partial-order plan and search for a solution. The input is the problem description, consisting of descriptions of the **initial state**, the **goal** and possible **actions**.

The problem can be interpreted as a search problem where the set of possible partial-order plans is the search space. The initial state would be the plan with the open preconditions equal to the goal conditions. The final state would be any plan with no open preconditions, i.e. a solution.

The initial state is the starting conditions, and can be thought of as the preconditions to the task at hand. For a task of setting the table, the initial state could be a clear table. The goal is simply the final action that needs to be accomplished, for example setting the table. The operators of the algorithm are the actions by which the task is accomplished.

Partial-order vs. total-order planning

Partial-order planning is the opposite of total-order planning, in which actions are sequenced all at once and for the entirety of the task at hand. That partial-order planning is superior to total-order planning, as it is faster and thus more efficient. They tested this theory using Korf's taxonomy of subgoal collections, in which they found that partial-order planning performs better because it produces more trivial serializability than total-order planning. Trivial serializability facilitates a planner's ability to perform quickly when dealing with goals that contain sub goals. Planners perform more slowly when dealing with laboriously serializable or nonserializable subgoals. The determining factor that makes a sub goal trivially or laboriously serializable is the search space of different plans. They found that partial-order planning is more adept at finding the quickest path, and is therefore the more efficient of these two main types of planning.