Arc consistency

Simplest form of propagation makes each arc consistent.
Can be run as a pre-processor or after each assignment: in Recursive-Backtracking after adding a new value an inference step is performed
– forward checking (removing values from the nodes constrained by the assigned var)
– arc consistency (starting with the arcs that connect the
assigned variable with unassigned ones)

PLANNING

State space: Each node represents a state of the real world and a transition is an addiction of an action in the action-sequence to reach a goal.
Plan space: One node represents a partial plan. The state of the search is a partial plan. When you go cross an arc you add an action from the action-set to the partial plan.

Possible method of searching for a plan:

- Progression: apply actions whose preconditions are satisfied until goal state is found or all states have been explored
- Regression: any positive effect of A that appears in G is deleted and each precondition literal of A is added unless it already appears.
- Heuristics search: it is very effective. “Relaxing the problem”, there are two possibilities:
  - Removing preconditions -> all the actions are completely free with any constraints.
  - Removing negated effects

Partial Ordered Plans (POP)

Partially ordered collection of actions with
- Start action has the initial state description as its effect
- Finish action has the goal description as its precondition
- temporal ordering between pairs of actions

Two additional elements are needed to characterize the planning process:

- Open precondition = precondition of an action not yet causally linked
- Causal links from outcome of one action to precondition of another

A *plan* is *complete* if every precondition is achieved. A *precondition* is *achieved* if: it is the effect of an earlier action and no possibly intervening action undoes it.
A clobberer is a potentially intervening action that destroys the condition achieved by a causal link. A conflict can be solved by adding:
- C ≺ A (demotion) or
- B ≺ C (promotion)

Nondeterministic algorithm: backtracks at choice points on failure:
– choice of action (Sadd) to achieve open precondition (Sneed)
– choice of demotion or promotion for clobberer

Selection of open precondition (Sneed) is irrevocable: the existence of a plan does not depend on the choice of the open preconditions.
POP is sound, and complete.
↳ soundness means that the algorithm does not yield any result that is untrue

General:
- number of open preconditions
-most constrained variable open preconditions that are satisfied in fewest ways
-a special data structure: the planning graph

Problem Specific:
Good heuristics can be derived from problem description (by the human operator). POP is particularly effective on problems with many loosely related subgoals

HIERARCHICAL TASK NETWORK PLANNING

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together, and state-of-the-art algorithms can generate solutions containing thousands of actions.
Bridging this gap requires planning at higher levels of abstraction.
We concentrate on the idea of hierarchical decomposition, an idea that pervades almost all attempts to manage complexity. The key benefit of hierarchical structure is that at each level of the hierarchy is reduced to a small number of activities at the next lower level, so the computational cost of finding the correct way to arrange those activities for the current problem is small.

HLA HIGH LEVEL ACTIONS

The basic formalism we adopt to understand hierarchical decomposition comes from the area of hierarchical task networks or HTN planning. We assume set of actions, now called primitive actions, with standard precondition–effect schemas. The key additional concept is the high-level action or HLA. Each HLA has one or more possible

refinements, into a sequence of actions, each of which may be an HLA or a primitive action. An HLA refinement that contains only primitive actions is called an implementation of the HLA. We can say, then, that a high-level plan achieves the goal from a given state if at least one of its implementations achieve the goal from that state. The "at least one" in this definition is crucial.
—not all implementations need to achieve the goal, because the agent gets to decide which implementation it will execute. Thus, the set of possible implementations in HTN planning —each of which may have a different outcome—is not the same as the set of possible. Outcomes in nondeterministic planning. There, we required that a plan work for all outcomes because the agent does not get to choose the outcome; nature does.

Describe HLA in HTN and provide example (referring to the previous problem)

A plan can be decomposed in a huge set of atomic primitive action which give only a small informative contribute for the entire plan or they introduce only a small progress with respect to the goal, so we introduce a high-level action.
An HLA is a sequence of primitive action directly runnable by an agent. in real planning problem. Wrt the given problem 'drop' or 'collect' could be a HLA because they are a sequence of movement or also the action go can be modelled in a sequence of primitive action like openCar, turn on Car... or takeBycicle turnLeft ecc ecc... .

Discuss the difference between state-space and plan-space planning. Describe the basic approach to plan space planning (POP)

State space is a transition from a state A to a state B will be caused by the addition of an action which will follow the action that had led to the state A. Instead in the plan space a transition to another node corresponds to inserting an action in the plan that has not to be strictly the following of the previous. In this representation actions can be in every point of the plan, without a sequential order.

The POP search procedure proceeds as follows:
The initial plan is composed only by the Start action and the Finish action, specifying that the start < finish (start comes before then finish)
Pick the open precondition p of an action B and look for an action A whose effect satisfies p.
Add the causal link A -> p -> B and specify the ordering A<B (A comes before then B). In addition to this if A is a new action add the orderings Start<A, B<Finish
Run the goal test which would succeeds if there are no more open preconditions.
Solve conflicts, otherwise backtrack.

Total-Order Planning (TOP)

Forward / Backward state-space searches are forms of totally ordered plan search: they explores linear sequences of actions that starts with the start state and ends with the goal state, maintaining a total ordering between all actions at every stage of the planning. This way we cannot take advantages of problem decomposition.

Partial-Order Planning (POP)

A partial-order plan specifies all actions that need to be taken, but specifies an ordering between them only where necessary.
This allows us to parallelize the computation (actions are often indipendent one from another) and we can even benefit from a situation in which we need to work on several subgoals indipendently: we can solve them with subplans and then put everything together.

↳ Flexibility in ordering the subplans

Example: putting on a pair of shoes

- Goal(RightShoeOn ^ LeftShoeOn)
- Init()
- Action: RightShoe
  - PRECOND: RightSockOn
  - EFFECT: RightShoeOn
- Action: RightSock
  - PRECOND: None
  - EFFECT: RightSockOn
- Action:LeftShoe
  - PRECOND: LeftSockOn
  - EFFECT: LeftShoeOn
- Action: LeftSock
  - PRECOND: None
  - EFFECT: LeftSockOn

Planning Summary

Author: **Nicolò Brandizzi**

Contributors:

SAPIENZA
UNIVERSITÀ DI ROMA

DIAG
Sapienza
October 2018

Contents

Abstract

This is **free** material! You should not spend money on it.
This notes are about the *Planning* part taught by professor Daniele Nardi in the Artificial Intelligence class. Everyone is welcome to contribute to this notes in any relevant form, just ask for a pull request and be patient. Remember to add your name under the contributors list in the title page when submitting some changes (if you feel like it).

Planning is another approach to search but instead of having a representation which is like a black box (specific for every problem and therefore it is not usable during the search process) we have a factor representation (it takes a specific form) and we can derive heuristics that are problem independent (general use).

factor representation: the representation is simply done by a set of variables with a finite domain of discrete values

Planning becomes effective over search when the state space becomes too large and complex.

Search vs Planning

They both do search. The key difference is that, while in search the representation is built ad hoc, with ad hoc data structures, to tackle a particular problem and actions are coded, in planning actions are specified by defining preconditions and effects.

Planning systems do the following:

1) open up action and goal representation to allow selection
2) divide-and-conquer by subgoaling
3) relax requirement for sequential construction of solutions

	Search	Planning
States	(Lisp) data structures	Fluents
Actions	(Lisp) code	Preconditions/Effects
Goal	(Lisp) code	Sets of states (conjunction)
Plan	Sequence from S_0	Constraints on actions

Factor representation can scale up: if we decide to introduce a new capability we just need to define a new action (add the action description) and the rest will be the same.

A fluent is a condition that can change over time.
Factor representation is a set of fluents.

Planning agents ≠ learning agents
= a set of techniques that start from a specific model of the problem based on the formalization of the state through a set of fluents. Planning algorithms can very efficiently solve problems by only looking at their structure, they don't need heuristics.

1 Classical Planning

A plan is a sequence of actions that allows us to reach a state where the goal condition holds, starting from the initial state.
For a classical planning problem we need to define 3 things:

- States, which are represented by *sets of instantiated literals* with a boolean value
  - Initial State
  - Goal State
- Actions or **action schema** composed of:
  - Action name
  - List of variables used in the schema
  - A precondition : define the state in which the action can be executed
  - An effect : result of an action
- An Environment:
  - Fully observable → the agent knows everything about the environment
  - Deterministic → the effect of each action is guaranteed
  - Static → while the agent is thinking, nothing happens (the world does not change if the agent does not make an action)
  - Discrete and Finite

Obviously an action can be executed in a state if the state *entails* the pre-conditions, i.e. the action is **applicable** in the state.

Lets consider the following example:

- Initial state: *Plane 1 in A*
- Goal state: *Plane 1 in B*
- Action: *Fly(Plane,From, To)*
  - Precondition : *Plane in From*
  - Effect: *Plane at To*

If we use the action *Fly(1,A,B)* the following thing happen:

- The plane is not in A anymore, so *At(1,A)* is deleted and added to the **delete list** [DEL(a)]
- The plane is in B, so now *At(1,B)* is true and is added to the **add list** [ADD(a)]

Intuitively we have that the precondition always refer to the time t while the effect refers to the time $t+1$.

Complexity of classical Planning We refer to:

- **PlanSAT** as the question to whenever it exist any plan that solves a planning problem, which is decidable for finite states, but becomes semi-decidable if we add function symbols to the language[1].

[1] The consequence is an infinite space

Everything in planning dates back to STRIPS

STRIPS : STanford Research Institute Problem Solver
= language for action description

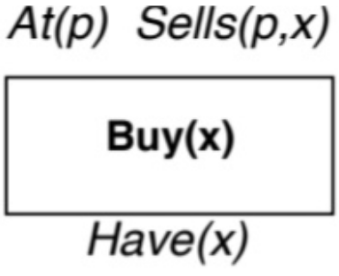
Fluents : expressions in the form $P(x_1, \dots, x_m)$ or $\neg P(x_1, \dots, x_m)$, where P is a predicate, $0 \leq m$ and x_i are either variables or constant symbols.

Fluents are typically used in order to characterize properties that change over time.
In simple terms, a fluent is a predicate that can have arguments and the arguments can be either variables or constants.
A fluent can be true or false based on the variables on which it depends

States are sets of instantiated fluents (only true, missing ones are supposed to be false) in the state we don't use negation : it is implicit.
↳ closed-world assumption

Action schema :

ACTION: $Buy(x)$
PRECONDITION: $At(p), Sells(p, x)$
EFFECT: $Have(x)$



Applicable action

An action a with preconditions q = Preconditions(a) is applicable to a state s if:

every positive fluent in q is in s
every negative fluent in q is not in s

why?

in the precondition we can have positive and negative fluents but in the state we can only have positive fluents and the others are supposed to be false.

Resulting (successor) state

we have to specify what is going to be the state after the execution of the action. The effects of the action can be positive fluents or negative fluents.

positive fluents have to be true in the state
→ ADD (a) (they have to be added)

negative fluents have to be false in the state
→ DEL (a) (they have to be removed if present)

ADD list = positive literals in the fluents
DEL list = negative literals in the fluents

All the fluents not in the ADD and DEL lists are kept unchanged
↳ persistence assumption

Everything not explicitely changed by effects persists, after the execution of an action.

Planning techniques

different models: the name is due to how we model the search for a solution (plan)

- search in state space = we can use these concepts to model a search problem like we did before with informed and uninformed search:
  - operators = actions = graph edges
  - states = set of fluents = graph nodes
  - ...

  only thing that's missing is the heuristic function

  we can search
  - forward (progression) initial state → goal
  - backward (regression) goal → initial state

  A state in our search for the plan will be a representation of the world

- search in plan space = change of problem representation:
  - state space: node = state in the world
  - plan space: node = partial plan

  Instead of considering a state of the search as a state of the world, we take a partial plan to be a state of the search: we start from a very simple partial plan (in most case this will be non executable) and we try to adapt it (add actions) until we eventually find the solution.

  A node is now a partial plan and a transition to another node corresponds to inserting an action in the plan.

- hierarchical planning = often the actions make very small changes with respect to the goal → tackle the problem at this level (primitive actions) can be very inefficient

  The operators of classical planning are preserved and a more abstract concept of action is introduced:

  high level actions (HLA)

  Planning proceeds by decomposing nonprimitive tasks recursively into smaller subtasks, until primitive tasks are obtained

  Action
  - Primitive action (A) → can be executed by the agent
  - High-level action (HLA) → cannot be executed but can be seen as a sequence of actions (Both A & HLA)

- **Bounded PlanSAT** as whenever there is a solution of length k, which remains decidable even with infinite states.

Forward state-space search Apply actions whose preconditions are satisfied unitl goal state is found or all states have been explored [2], with the following consequences:

- $Results(s, a) = (s - DEL(a)) \cup ADD(a)$
- The exploring of irrelevant actions
- The issue for the large space this kind of problems have.
- Strong domain-independent heuristics can be derived automatically.

Backward relevant-state search If a domain can be expressed in PDDL then we can do regression on it [3]. If we want to go from the goal g with an action a to the state g' we have: backwards

$$g' = (g - ADD(a)) \cup Precond(a)$$ → reverse the fluents that are in the preconditions of a

We can do this as long as an action a is **relevant** to the state g. Regression keeps the branching factor low, but it makes harder to come up with good heuristic, thus forward is preferred.

consistent actions: they do not destroy any of the fluents in the goal when trying to invert them

Heuristic for Planning Remember that an admissible heuristic can be derived from the **relaxed problem** which is easier to solve.
Search problems can be seen as a graph where nodes are stated and edges are actions. There are three ways you can relax a problem with graph:

- **Ignoring preconditions**: drop all precondition of an action. Usually this means that the number of actions to solve the problem is the same as the number of unsolved goal. On the other hand if we may have actions which achieve multiple goals or actions which can undo other actions [4]. .
- **Ignore delete list**: remove all negative literals from effects [5], monotonic progression towards the goal.
- **State abstraction**: many-to-one mapping where we decrease the number of states by ignoring some fluent.
- **Decomposition**: dividing the problems into parts and solving them independently [6], but it is a pessimistic approach [7] when sub-plans contains redundant actions.

Lemma If the effect of P_i leaves all the preconditions of P_j unchanged, then the estimated cost $Cost(P_i) + Cost(P_j)$ is admissible and more accurate of the maximum.

[2] Mapping planning into a search problem
[3] This is because we keep in memory both the DEL(a) and the ADD(a) lists
[4] Can be ignored to relax the problem
[5] No action can undo the progress made by another action.
[6] Assume sub-goal independence
[7] Not admissible

To find a suitable heuristics we can decompose the problem into n smaller, individual problems; each one of them will have a cost; which cost should we use for the heuristic? Take the max or their sum?
Better do the max because maybe solving one subgoal can help solving others subgoals: summing will return

a higher cost in this case, hence the heuristic is not admissible.

Heuristic is admissible if it does not overestimate the cost.

max ⟶ always admissible

sum ⟶ not always admissible

* $g' = (g - ADD(a)) \cup Precond(a)$

remove from the current state the fluents that have been added by the chosen action

introduce the preconditions: apply the actions in Precondition (a)

$$POS(g') = (POS(g) - ADD(a)) \cup POS(PREC(a))$$
$$NEG(g') = (NEG(g) - DEL(a)) \cup NEG(PREC(a))$$

long sequences of actions are unrealistic in robotics, so we can decompose them into partial (shorter) sequences.

2 Partial Order Planning [POP]

Many sub-problems are independent so we can work on several sub-goals independently, solve them and then combine them to achieve a solution to the original plan.
A Partial Order Planner is a planning algorithm which can place two action into a plan without specifying which comes first, so that the solution is in the form of a *graph* rather than a sequence. POPs are created by a search in the *space of plans* rather than through the state space, where an action is actually a plan [8]. They work by **fixing flaws**[9] in the plan with the principle of **least commitment** [10].

Each plan has four components:

- A **set of actions** that makes up the steps for the plan. An empty plan has only the *Start* and *Finish* actions.
- A set of **ordering constraints** between pairs of actions in the form $A \prec B$ (A happens some time before B).
- **Causal links** $A \xrightarrow{p} B$ (A achieve p for B) [11]. The presence of causal links lead to *early pruning* of portions of state space that, because of not resolvable conflicts, contain no solutions.
- **Open preconditions** for an action not yet causally linked, i.e. a precondition that is not achieved by any action yet.

If a plan has the following proprieties than it is defined as **consistent**:

- All the open preconditions are achieved [12]
- There are no **cycles**, i.e. $A \prec B$ and $B \prec A$ which is also a contradiction.
- There are no **conflicts**, that is when $A \xrightarrow{p} B$ and an effect of another action C is $\neg p$ where $A \prec C$ and $C \prec B$. Note that a conflict can be solved by applying:
  - A **Demotion** $C \prec A$ = put before first
  - A **Promotion** $B \prec C$ = put after last

A consistent plan with no open preconditions is a **solution**.

The algorithm works in the following way:

1. The plan only contains *Start* and *Finish* with $Start \prec Finish$ → Starting point (plan)
2. The successor function: → Successor function

[8] The difference is subtle. An action may be *Move(what,where)* where a plan is *Move(ObjA,Pos1)*.
[9] A flaw is anything that keeps the partial plan from being a solution
[10] Delaying a choice during the search until it is strictly necessary.
[11] For example when putting on shoes we may have $RightSock \xrightarrow{RightSockOn} RightShoe$, meaning that putting the right sock on achieves the condition necessary for putting the right shoe on.
[12] A precondition is *achieved* if no action can undo it's effects.

→ Termination condition

(a) pick an open condition p of an action B
(b) pick an action A that achieves p
(c) add the causal link $A \xrightarrow{p} B$ and $A \prec B$
(d) resolves conflicts if possible, otherwise backtrack

3. The goal test succeeds when there are no more open preconditions

POP is sound and complete

if the algorithm finds a solution, then the solution is correct

if there is a solution, the algorithm will find it

POP: choice a suitable action that will fulfill the open preconditions present and iterate this procedure backwards from the goal state to the start state

POP heuristics: number of open preconditions
→ there may be cases where one action achieves more than one preconditions and it is no more assured in this case that the number of open preconditions does not overestimate the number of steps: the admissibility can be lost.

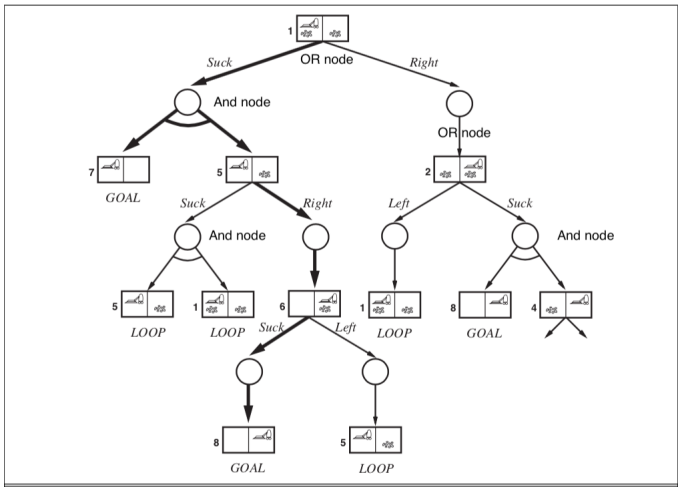
3 Non-Deterministic domains

3.1 Searching with Non-deterministic actions

When the future percepts cannot be determined in advance [13] the solution to a problem is a **contingency plan**. In this case the *result* of a problem is a function that return a **set** of possible outcomes states [14]. That is why the solution of a non-deterministic problem can contain nested **if-then-else** statements so that the form is *tree-like* rather than a sequence.

And-or trees The branching of a tree can depend on:

- The **agent's choices**, which will generate OR nodes
- The **environment's choice**, which will generate an AND node.



Non-cyclic Solutions The issue is that cycles often arise in this kind of problem.We can consider a solution in which: if the current state is identical to a state on the path from the root, than return false. This method guarantee to terminate in every finite space and return a solution which is **non-cyclic**.

Cyclic Solutions When we deal with non-deterministic actions we can get to the point where an action must be repeated some times to make it work [15]. So we want to keep doing an action until some state is reached [16]. This leads to **cyclic solution** which will eventually reach the goal provided that each outcome of a non-deterministic action eventually occurs.

[13] The state is only partially observable.
[14] Rather than a specific one.
[15] Slippery floor for robot movement
[16] Using *while* conditions

3.2 Searching with partial observation

For this kind of searching we need to introduce the concept of **belief state**, representing the agent's current belief about the possible physical states it might be in.

Sensor less belief With a specific sequence of action an agent can **coerce** the world into a state X [17]. Since the agent always know its own belief state we can derive that the latter is *fully observable*, furthermore the solution (if any) is always a sequence of actions.
A sensor-less problem [P] has the following elements:

- **Belief states**: contains every possible set of physical states
- **Initial state**: the set of all states in P
- **Actions**: there are two cases which depends on their effect on the environment
  - *Illegal action have no effect on the environment*: then we can take the **union** of all the actions in the current belief state
  - *Illegal action have effect on the environment*: then we take the **intersection**
- **Transition model**: The model which generate a new belief state b' [18] from a belief b troughs an action a, it depends on the type of action:
  - *Deterministic* action: $b' = Result(b, a) = \{s' : s' = Result_P(s, a), s \in b\}$
  - *Non-Deterministic* action: $b' = Result(b, a) = \{s' : s' = Results_P(s, a), s \in b\} = \cup_{s \in b} Reulsts_P(s, a)$
- **Goal test**: the belief state satisfies the goal only if all the physical states in it satisfy the $GoalTest_P$
- **Path cost**: assumed same for every action

Since The size of each belief state is exponential we can use the **incremental belief-state search** in which we first find a solution that works for state 1; then check if it works for state 2; if not, go back and find a different solution for 1, and so on.

Searching with observations In this case we have the *Percept(s)* function which returns the percept in a given state. When an observation is *partial* then we could have it generated by several states. In this case the difference with the sensor-less problem mentioned above are the following:

- There is a **prediction** stage.
- There is a **observation prediction** stage in which we determine the set of percepts that could be obtained in the predicted belief state
- The **update** stage, for each possible percepts we get the belief state that would result from it [19].

[17] That is the agent is sure to be in X after execution a sequence of actions
[18] Also called the *prediction* step.
[19] Reduces uncertainty.

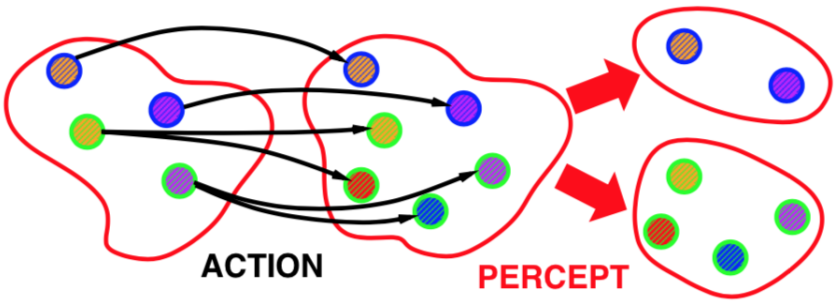
3.3 Planning in Non-Deterministic domains

For planning we augment the PDDL with a **precept schema**.
Online planning can generate a contingent plan with fewer branches at first, and deals with problems as they arise. While **re-planninig** agents would check the result and replan in case of unexpected failure.

Sensorless planning We can convert a sensorless problem to a belief-state planning problem where the transition model is represented by a set of action schemas and the belief model is represented by a logical formula [20].
If an effect depend on the state then we have a **conditional effect** which is denoted like this:

$$\textbf{when} \quad condition : effect$$

This kind of effect brings the belief state to have an exponential size in the worst case.



The different between unsatisfied conditional effect and unsatisfied pre-condition is that the latter do not permit the action to occur, thus the resulting state is **undefined**,while an unsatisfied conditional effect **does not change the state**.
Other kind of solutions can be:

- Perform some actions which keeps the belief state as simple as possible [21] and lower uncertainty.
- Not compute the belief state after some actions, but rather represent it as being followed by those action without computing it.

The final piece is to implement a good heuristic function to guide the search. Since any subset of the original belief is easier to solve we have that any heuristic which is admissible for a subset of the original belief state is also admissible for the entire belief state with the maximum function:

$$H(b) = max(h_1(b), ..., h_n(b))$$

[20] Instead of being all the possible enumeration of the states.
[21] Like when we check the time every once in a while to lower our uncertainty of the current time.

3.4 Online replanning

Execution monitoring Determine the need for a new plan. Some branches of a partially constructed contingent plan can simply say Replan; if such a branch is reached during execution, the agent reverts to planning mode. Replanning may occurs in the following situations:

- **Missing precondition**
- **Missing effect**
- **Missing state variable**
- **Exogenous events**: changes in the goal

Action Monitoring Before executing an action, the agent checks if all the preconditions still hold.

Plan Monitoring Before executing an action, the agent verifies that the remaining problem will still succeed. It can detect failure by checking the preconditions for success of the entire remaining plan. It cut off the execution of a dead plan as soon as possible and it also allows **serendipity** [22].

Goal Monitoring Before executing an action, the agent checks if there is a better set of goals it could try to achieve.

[22] Accidental success.

4 High-level actions

We introduce the concept of **high-level action** [HLA] which is part of the **hierarchical task networks** [HTN] which assumes the full *observability*, *determinism* and *availability* of a set of actions called **primitive actions**.
A **refinement** is a sequence of actions which describes [23] an HLA. If a refinement contains only primitive actions then it is called **implementation**.
A high-level plan achieves the goal from a given states if at least one of its implementations achieves the goal from that state.
A set of possible implementations in HTN planning is not the same as a set of possible outcomes in non-deterministic planning.

4.1 Searching for primitive solutions

It always start with a top level action called **Act**, then, for each primitive action a_i, we provide a refinement of Act with steps $[a_i, Act]$. The algorithm then becomes one that repeatedly chooses an HLA in the current plan and replace it with one of its refinements until the goal is achieved. This is computationally expensive, given:

- d primitive actions
- b allowable actions at each state
- k actions
- r possible refinements

The possible decomposition trees are:

$$r^{\frac{d-1}{k-1}}$$

Which, by taking small r and large k [24] , results in huge savings.
Some algorithms may even save successful plans in memory to later build newer plans on top of the older ones, leading to a more competent planner over time [25].

4.2 Searching for abstract solutions

If we write precondition-effect description for an HLA we can check whenever an HLA achieve goal without decomposing it with the refinements. So an HLP must have at least one implementation that achieves the goal to be considered doable, this is called the **downward refinement property**.
The problem lays in the description of the effect of an action which can be implemented in many different ways. A solution might be to include only the positive effects that are achieved for *every* implementation of the HLA and the negative effects of *any* implementation.
We introduce the notion of **reachable set** for an HLA h [26] in a state s, the

[23] Or decomposes.
[24] Small number of refinements each yielding a large number of actions.
[25] The problem lays in generalizing the methods that are constructed.
[26] Written Reach(s,h)

set of states reachable by any implementation of HLA. We can now say that a sequence of HLAs achieves the goal if its reachable set intersect the set of goal states.

Angelic semantics Given a *fluent* [27], a primitive action can add, delete or leave unchanged a fluent. The HLA under angelic semantics can control the fluent's value depending on which implementation is chosen.
For example to HLA *Go(home,work)* with two possible refinements:

- Drive(home,work)
- Call(taxi),Taxi(home,work)

can have as requirement for the $Taxi$ action to have $Cash$. Under angelic semantics the agent can choose to delete $Cash$.
This results in the derivable descriptions of HLAs from their refinements.
The angelic approach can be extended to find the least-cost solution by introducing the most efficient way to get to a state.

Descriptions There are two kind of descriptions which are kind of a upper/lower bound for the reachable set:

- **Optimistic**: may overstate the reachable set of an HLA
- **Pessimistic** : may understate the reachable set.

We can derive that if an optimistic reachable set does not intersect the goal then the plan does not work, on the other hand if a pessimistic description set intersect the goal then the plan works for sure. Additional refinement is needed when the optimistic intersects but the pessimistic doesn't.

[27] A state varaible