PLANNING BASED ON GRAPH

- In 1995 a new concept of planner based on graph has been proposed by Blum and Furst CMU – GRAPHPLAN

- While planning it creates a graph called **Planning Graph**
  - At each step of the search this data structure is extended

Here we have a single data structure, graph plan, and for the first time it indroduces the notion of Time

It return the shortest possible plan

- Graphplan inserts the TIME dimension in the plan construction process.

- It is a correct and complete planner among the most efficient that have been built

GRAPH PLAN: features

- Graphplan uses the CLOSED WORLD ASSUMPTION falling into the category of off-line planners
- Graphplan returns either the shortest possible plan or returns an inconsistency.
- Graphplan Inherits from linear planners the EARLY COMMITMENT feature: ex. the action A is executed at time step 2
- Graphplan inherits from non-linear partial order planners the ability to create partially ordered sets of actions:
- It generates *parallel plans*

GRAPH PLAN

- Actions are represented as the ones in STRIPS
  - PRECONDITIONS
  - ADD LIST
  - DELETE LIST
- Objects have a type
- There is an action **no-op** that does not change the state (frame problem)
- States are represented as sets of predicates that are true in A given state.

PLANNING GRAPH

- The planning graph is a **directed leveled graph**
  - nodes belong to different levels
  - arcs connect nodes in adjacent levels.

T0 | T1 | T2 Different levels

- Level 0 corresponds to the initial state
- In the planning graph proposition levels and action levels are interleaved and correspond to increasing time steps

- In the planning graph interfearing actions and propositions in a time step t can appear.

PLANNING GRAPH

- In a **planning-graph** There are different levels
  - Proposition level: nodes represent propositions
  - Action level: nodes represent actions

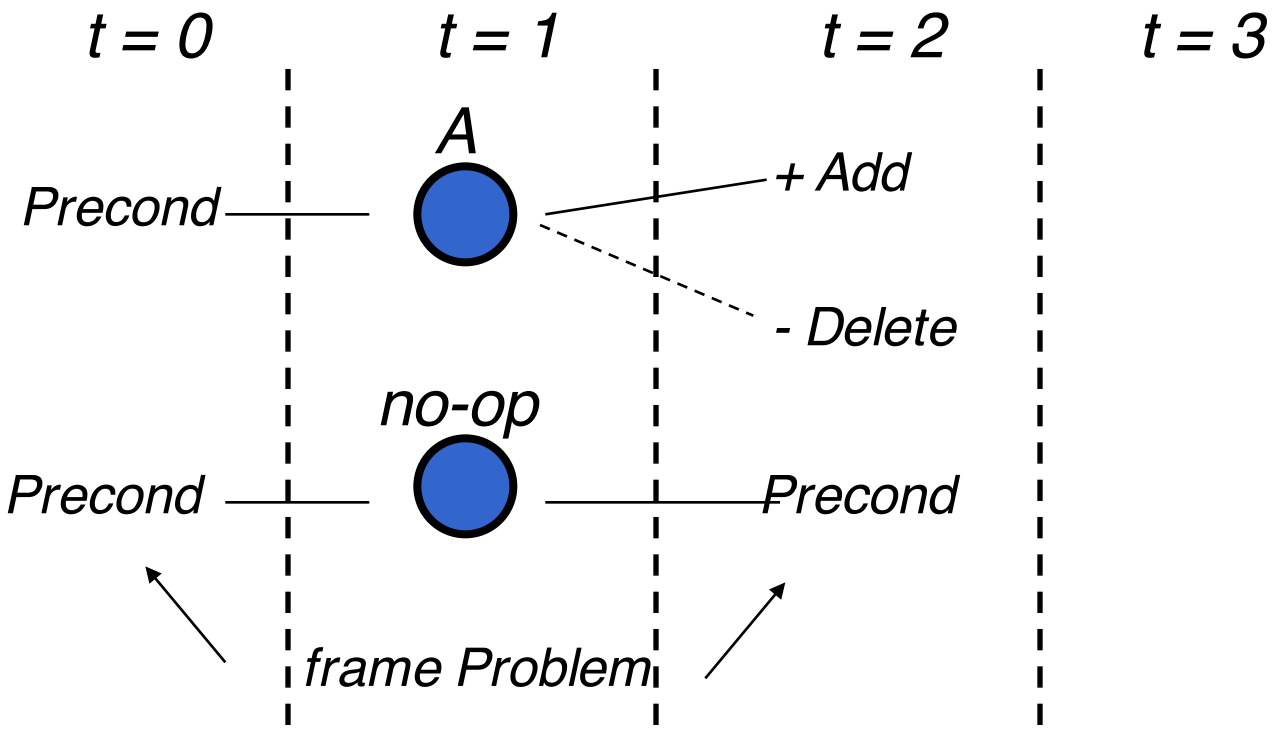
- Level 0 corresponds to the initial state and it is a proposition level

T1 | T2 | T3 | T4

propo. level | action level | propo level | action level

- Arcs are divided into:
  - Precondition arcs (proposition → action)
  - Add arcs (action → proposition)
  - Delete arcs (action → proposition)

PLANNING GRAPH

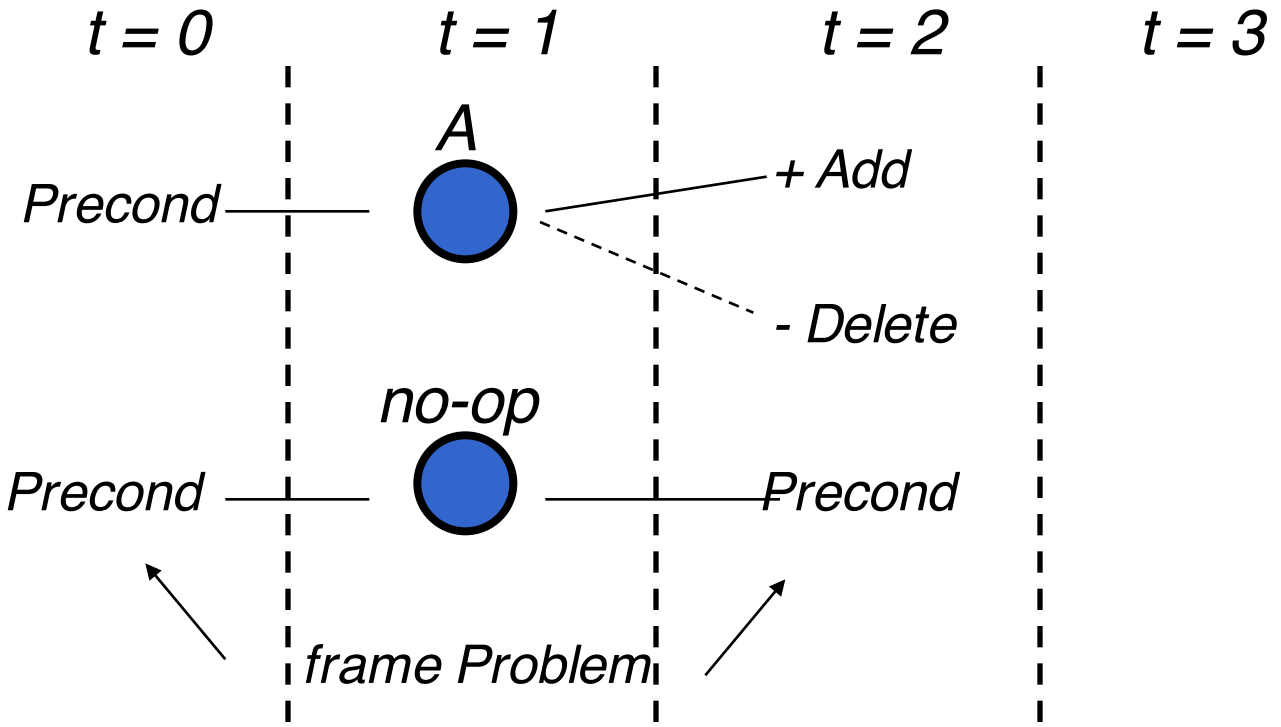


In each time step an action A can be inserted if in the previous time step all preconditions of A exist

when we arrive at the level that all the preposition (all true and not incompatible) of our goal are present, we start to go backwarw.

PLANNING GRAPH

in the layer action we put all the action that are posibile to do, thanks to the precondition given at t-1, we do not care if they are incompatible



There are special actions representing the “doing nothing” activity.

These actions are called no-op or frame actions

PLANNING GRAPH

- Each action level contains:
  - all actions that are applicable at that time step
  - constraints connecting pairs of actions that cannot be performed simultaneously

- Each proposition level contains all literal that might result from any choice of actions in the previous time step including no-op

- **NOTE**: the construction process of the planning graph does not imply ANY CHOICE on the selection of the action that will be inserted in the plan

wHILE BUILIND THE PLANNING GRAPH WE ARE NOT DECIDING WHICH ACTION WILL BE EXECUTED IN THE SOLUTION

INCONSISTENCIES

During the construction of the planning graph inconsistencies are identified, in particular

- Two **actions** can be inconsistent in the same time step
- Two **propositions** can be inconsistent in the same time step

In this case the action / propositions are **mutually exclusive**

- They can not appear together in a plan
- **But they may appear** in the same level **of the planning graph**

INCONSISTENT ACTIONS

- **Inconsistent effects**: one action negates the effect of another
  - The move action (part, dest) has the effect not at (part) while the no-op action on at (part) has this effect
- **Interference**: an action deletes a precondition of the other
  - The move action (part, dest) has the effect not at (part) while the no-op action on at (part) has this as a precondition
- **Competing needs**: two actions that have mutually exclusive preconditions.
  - The load share (load means) has as a precondition not in (load means) while the action unload (load means) has as a precondition for (load means)

INCONSISTENT PROPOSITIONS

- Two propositions are inconsistent if
  - one is the negation of the other
  - If all the ways to reach them are mutually exclusive

- Furthermore, there might be domain dependent inconsistencies
  - Ex: an object cannot be in two places at the same time in the same time step.

PLANNING GRAPH: ALGORITHM

- The planning graph is built as follows:
  - All true propositions in the initial state are inserted in the first *proposition level*

  CREATION of Action LEVEL

  - For every operator and every way to unify its preconditions to propositions in the previous proposition level, enter an action node IF two propositions are not labeled as **mutually exclusive**
  - In addition, for every proposition in the previous proposition level, add a no-op operator
  - Check if the action nodes do not interfere each other otherwise mark them as mutually exclusive

PLANNING GRAPH: ALGORITHM

CREATION OF PROPOSITION LEVEL

- For each action node in the previous level, add propositions in its add list through solid arcs and add dotted arcs connected to the propositions in the delete list
- Do the same process for the no-op operators
- Mark as mutually exclusive two propositions such that all the ways to achieve the first are incompatible with all the ways to reach the second.

PLANNING GRAPH: ALGORITHM

Algorithm for the construction of the planning graph:

1. **INITIALIZATION:**
   All true propositions in the initial state are included in the first proposition level
2. **CREATING AN ACTION LEVEL:**
   1. ∀operator ∀every way to unify its preconditions to not mutually exclusive propositions in the previous level, enter an action node
   2. For every proposition in the previous proposition level, enter a no-op operator
   3. Identify the mutual exclusive relationship between the newly constructed operators

PLANNING GRAPH: ALGORITHM

3. **CREATING A PROPOSITION LEVEL:**
   1. For each action node in the previous action level, add the propositions in his add list through solid arcs
   2. For every "no-op" in the previous level, add the corresponding proposition
   3. For each action node in the previous action level, link propositions in his delete list by dashed arcs
   4. Identify incompatible propositions.

EXAMPLE

- We have a cart R and two loads A and B that are in the starting position L and must be moved to the target position P.

- Three actions (we see the syntax later)
  - MOVE (R, PosA, PosB)
  - LOAD (Pos, Object)
  - UNLOAD (Pos, Object)

EXAMPLE

- MOVE(R,PosA, PosB)
  - PRECONDIZIONI: at(R,PosA), div(PosA, PosB), hasFuel(R)
  - ADD LIST: at(R,PosB)
  - DELETE LIST: at(R,PosA), hasFuel(R)

- LOAD(Object, Pos)
  - PRECONDIZIONI: at(R,Pos), at(Object,Pos)
  - ADD LIST: in(R, Object)
  - DELETE LIST: at(Object,Pos)

- UNLOAD(Object,Pos)
  - PRECONDIZIONI: in(R, Object), at(R,Pos)
  - ADD LIST: at(Object,Pos)
  - DELETE LIST: in(R, Object)

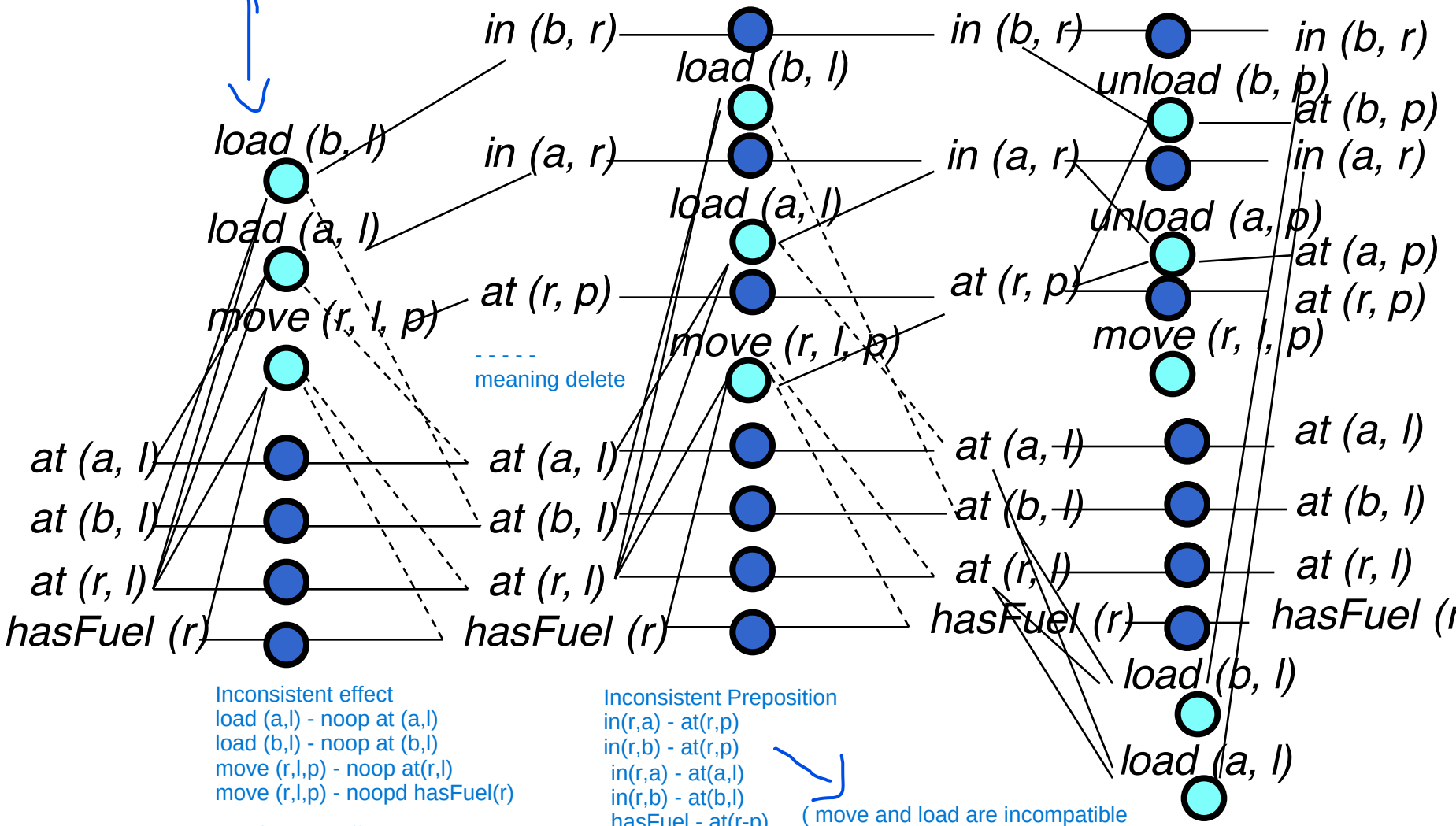
EXAMPLE

- Objects
  - Cart r
  - Objects a, b
  - Locations l, p
- Initial State
  - at(a,l)
  - at(b,l)
  - at(r,l)
  - hasFuel(r)
- Goal:
  - at(a,p)
  - at(b,p)

PLANNING GRAPH

This all the action that we can do at the start from the initial situation

At the exam we will have to write only 2 levels. (Level 0 is given)



Inconsistent effect
load (a,l) - noop at (a,l)
load (b,l) - noop at (b,l)
move (r,l,p) - noop at(r,l)
move (r,l,p) - noopd hasFuel(r)

Interference effect
move(r,l,p) - load(a,l)
move(rl,p) - load (b,l)

Inconsistent Preposition
in(r,a) - at(r,p)
in(r,b) - at(r,p)
in(r,a) - at(a,l)
in(r,b) - at(b,l)
hasFuel - at(r-p)
at(r-p) - at(r,l)

(move and load are incompatible so also their next preposition)
if an action add a propositiona and it delete another one these 2 are incompatible

EXTRACTION OF A VALID PLAN

- Once the planning graph is built, we have to extract a **valid-plan, i.e.,** a connected and consistent subgraph of the planning graph

Features valid plan

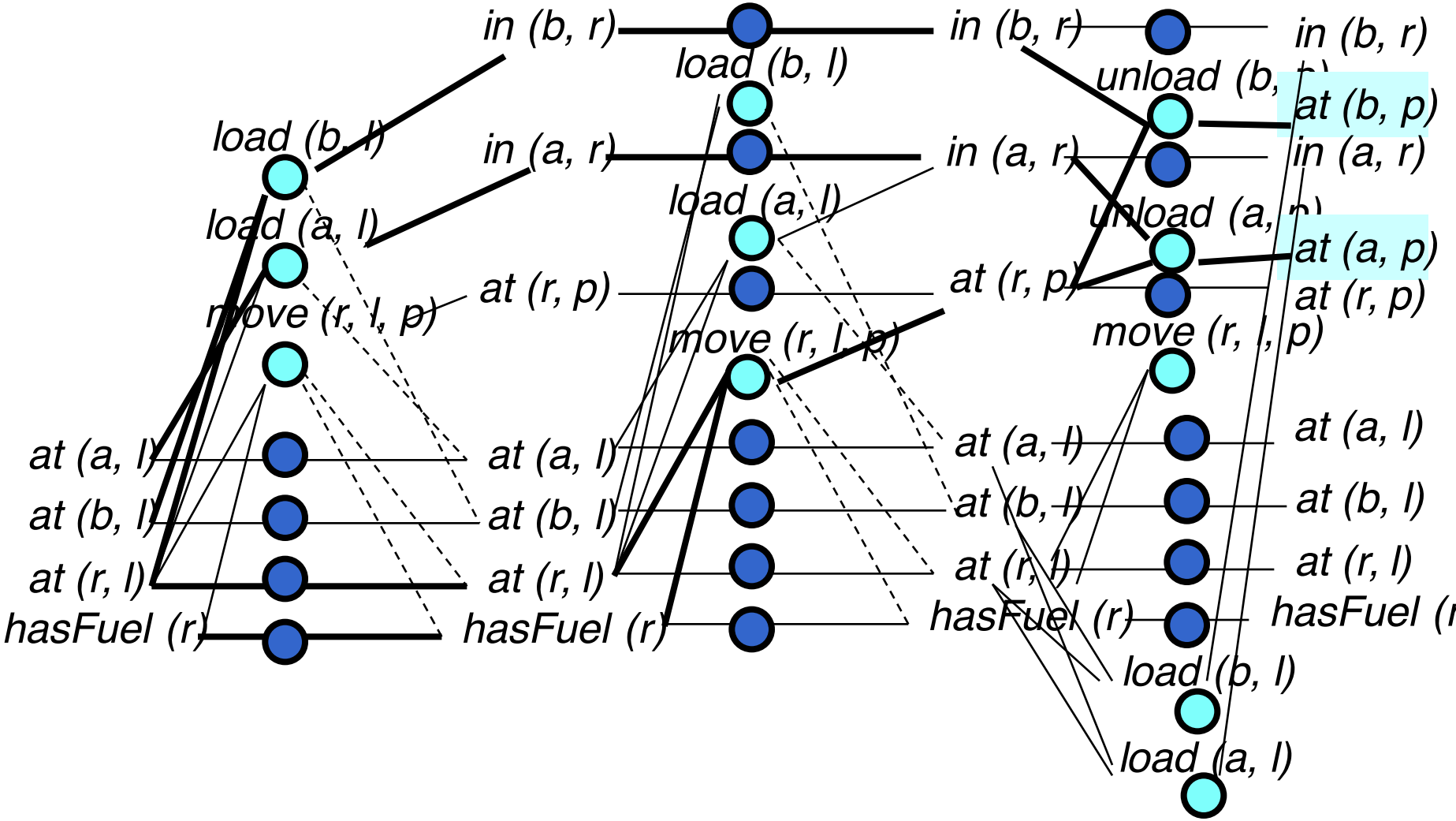
- Actions in the same time step can be performed in any order (do not interfere)
- Propositions at the same time step are not mutually exclusive
- The last time step contains all the literals of the goal and these are not marked as mutually exclusive

THEOREMS

1. If there is a valid plan then this is a subgraph of the planning graph.
2. In a planning graph two actions are mutually exclusive in a time step if a valid plan that contains both does not exist .
3. In a planning graph two propositions are mutually exclusive in a time step if they are inconsistent, ie one of them denies the occurrence of the other.

Important consequence:
The inconsistencies found by the algorithm prune paths in the search tree

VALID PLAN


in (b, r)
load (b, l)
in (b, r)
in (b, r)
load (b, l)
in (a, r)
in (a, r)
load (a, l)
load (a, l)
at (r, p)
at (r, p)
move (r, l, p)
move (r, l, p)
at (a, l)
at (a, l)
at (b, l)
at (r, l)
hasFuel (r)
at (a, l)
at (b, l)
at (r, l)
hasFuel (r)
unload (b,
at (b, p)
in (a, r)
unload (a,
at (a, p)
at (r, p)
move (r, l, p)
at (a, l)
at (b, l)
at (r, l)
hasFuel (r)
load (b, l)
load (a, l)

ALGORITHM

```
function GRAPHPLAN(problem):
         graph = GRAFO_INIZIALE (problem)
         targets = GOAL (problem)
         do loop:
            if objectives not mutex last step:
               Sol = ESTRAI_ SOLUTION (graph, objectives)
               if Sol ≠ fail: return Sol
               else if LEVEL_OFF (graph): return fail
            = ESPANDI_GRAFO graph (graph, problem)
```

- The first node contains the initial planning graph. This contains only one time step (proposition level) with true propositions in the initial state.
  The initial graph is extracted from `GRAFO_INIZIALE (problem)`

ALGORITHM

- The goal to reach is extracted from the function `GOAL(problem)`
- If goals are not mutually exclusive in the last level, then the planning graph **could** include a valid plan. The valid plan is extracted through BACKWARD search `ESTRAI_SOLUZIONE(graph, objectives)` that provides either a solution or a failure
  - Proceed level by level to better exploit the mutual exclusion constraints
  - recursive method: given a set of goals at time t the algorithms looks for a set of actions at time t-1 who have such goals as add effects. **The actions should not be mutually exclusive.**
  - The search is the hybrid breadth/depth first and complete

ALGORITHM

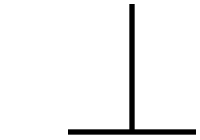
- **memoization** (Not a typo!)
  If at some step of the search, a subset of goals is not satisfiable, graphplan saves this result in a hash table. Whenever the same subset of goals is selected in the future will automatically fail

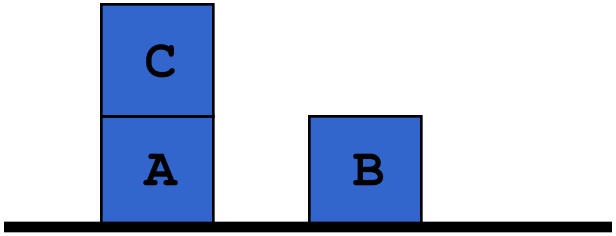
FIRST EXAMPLE: Block World

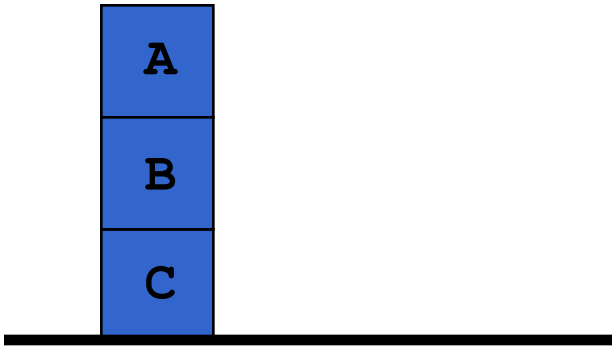
```
(BlockA OBJECT)
(BlockB OBJECT)
(BlockC OBJECT)

(preconds
     (On-table blockA)
     (On-table blockB)
     (On blockC blockA)
     (Clear blockB)
     (Clear blockC)
     (Arm-empty))

(effects
     (On blockA blockB)
     (On blockB blockC))
```







Place them in block_facts file

FIRST EXAMPLE: Block World

Place them in block_ops file

```
(operators
  PICK-UP
  (Params (<ob1> OBJECT))
  (preconds
    (Clear <ob1>) (on-table <ob1>) (arm-empty))
  (effects
    (Holding <ob1>)))

(operators
  PUT-DOWN
  (Params (<b> OBJECT))
  (preconds
    (Holding <b>))
  (effects
     (Clear <b>) (arm-empty) (on-table <b>)))
```

FIRST EXAMPLE: Block World

```
(operators
 STACK
 (Params (<b> OBJECT) (<underob> OBJECT))
 (Preconds (clear <underob>) (holding <b>))
 (Effects (arm-empty) (clear <b>)
            (On <b> <underob>)))

(operators
 UNSTACK
 (Params (<b> OBJECT) (<underob> OBJECT))
 (Preconds (on <b> <underob>) (clear <b>)
            (Arm-empty))
 (Effects (holding <b>) (clear <underob>)))
```

FAST FORWARD

Fast Forward

FF is a extremely efficient heuristic planner introduced by Hoffmann in 2000

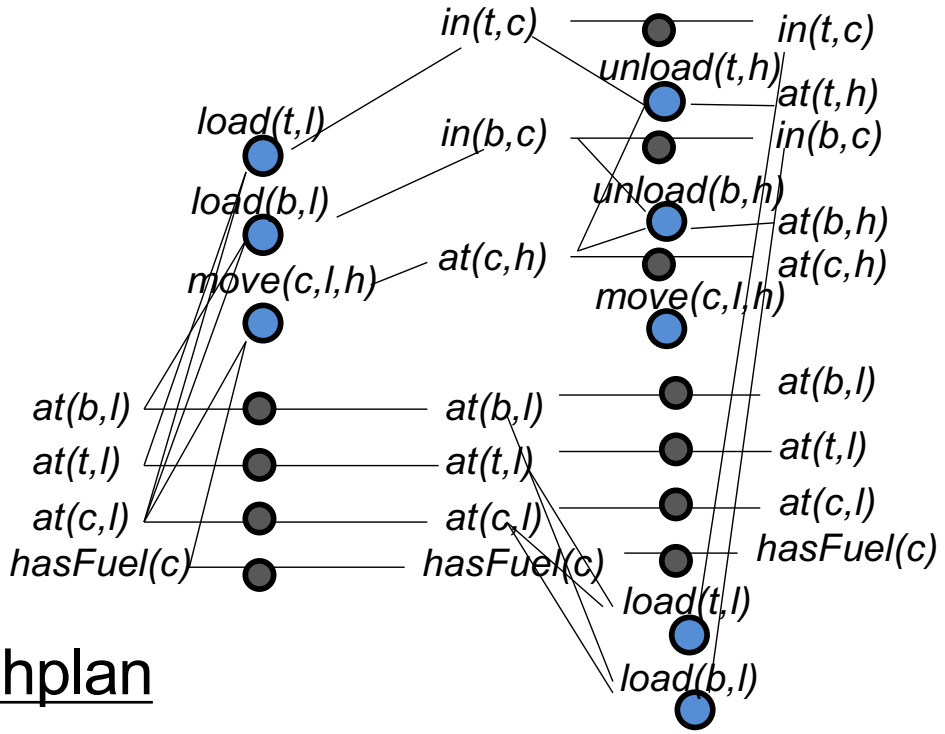
- *Heuristic = in each state S we compute an estimate of the distance to the goal*

Basic operation: hill climbing + A *

1. From a state S, examine all successors S '
2. If a successor state S* exists better than S, move on it and go back to point 1
3. If there is no state with a better evaluation, a complete A* search is run, using the same heuristic

HEURISTIC FUNCTION

- Given a problem P, a state S and a goal G, FF considers a relaxed problem P+ that you get from P by neglecting delete effects actions



- FF solves P+ with graphplan
- The number of actions in the resulting plan it is used as heuristics

HEURISTIC FUNCTION

FF actually uses a so-called "enforced hill climbing"

```
function FF(problem): Return solution or fails
         S = Initial_state (problem)
         k = 1
         do loop:
            explore all states S 'at k steps
            if a Better state S* is found: S = S *
            else if k can be increased: k = k + 1
            else perform complete A* search
```

- In practice it is a complete breadth first search
- A solution is always found, unless the current one is not a dead end

REFERENCES

- GRAPHPLAN
  - http://www.cs.cmu.edu/~avrim/graphplan.html
  - A. M. Blum and Furst, "Fast Planning Planning Through Graph Analysis", Artificial Intelligence, 90: 281-300 (1997).
- Fast Forward
  - http://members.deri.at/~joergh/ff.html
  - J. Hoffmann, "FF: The Fast-Forward Planning System", in: AI Magazine, Volume 22, Number 3, 2001, Pages 57-62
- Blackbox FOR PROJECTS
  - http://www.cs.rochester.edu/u/kautz/satplan/blackbox/index.html
  - SATPLAN: http://www.cs.rochester.edu/u/kautz/satplan/index.htm
  - Henry Kautz and Bart Selman, "Planning as Satisfiability", Proceedings ECAI-92.
  - Henry Kautz and Bart Selman, "Unifying SAT-based and Graph-based Planning", Proc. IJCAI-99, Stockholm, 1999.