CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

Schedule

- 10/31: Continue planning (HW3 out)
- 11/2: Finish planning, start probability (Bayesian networks)
- 11/6: Withdrawal deadline
- 11/7: TA will go over HW2
- 11/9: Continue probability (Bayesian networks, Markov models)
- 11/14: Markov decision processes (HW4 out)
- 11/16: Reinforcement learning
- 11/21, 11/28, 11/30, 12/5: Machine learning (classification, regression, clustering, deep learning)
- 12/7: Project presentations and class project due
  - Project code likely due bit earlier, 12/2-12/4, TBA.
- Final exam on 12/14

Announcements

- HW3 out today due 11/14 (2:05pm in lecture or 2:00pm on Moodle)
  - https://www.cs.cmu.edu/~sganzfri/HW3_AI.pdf
  - Must be done individually (no partner)
- Midterm exams
- HW2 solutions and graded assignments
- Midterm grades and withdrawal deadline

Class project

- For the class project students will implement an agent for 3-player Kuhn poker. This is a simple, yet interesting and nontrivial, variant of poker that has appeared in the AAAI Annual Computer Poker Competition. The grade will be partially based on performance against the other agents in a class-wide competition, as well as final reports and presentations describing the approaches used. Students can work alone or in groups of 2.
- Link to play against optimal strategy for one-card poker:
  - http://www.cs.cmu.edu/~ggordon/poker/
- Paper on Nash equilibrium strategies for 3-player Kuhn poker
  - http://poker.cs.ualberta.ca/publications/AAMAS13-3pkuhn.pdf

Linear programming (LP)

- Countless real-world applications have been successfully modeled and solved using LP techniques. This has produced an ongoing revolution in the way decisions are made throughout all sectors of the economy. Typical applications include the scheduling of airline crews, the distribution of products through a manufacturing supply chain, and production planning in the petrochemical industry.
- Because of the simplicity of the LP model, software has been developed that is capable of solving problems containing millions of variables and tens of thousands of constraints. Computer implementations are widely available for most mainframes, workstations, and microcomputers. A variety of problems with nonlinear functions, multiple objectives, uncertainties, or multiple decision makers, such as those arising in game theory, can be modeled as linear programs.

LP solution concepts

- **Solution**: An assignment of values to the decision variables is a solution to the LP model. Given a solution, the expressions describing the objective function and the constraints can be evaluated. A solution is *feasible* if all the constraints, the non-negativity restrictions, and the simple upper bounds are satisfied. If any one of the restrictions is violated, the solution is *infeasible*.
- **Optimal solution**: A feasible solution that maximizes or minimizes the objective function (depending on the criterion). The purpose of an LP algorithm is to find the optimal solution or to determine that no feasible solution exists.

LP solution concepts

- **Alternative optima**: If there is more than one optimal solution (solutions that yield the same value of the objective z), the model is said to have multiple or alternative optimal solutions. Many practical problems have alternative optima.
- **No feasible solution**: If there is no specification of values for the decision variables that satisfies all the constraints, the problem is said to have no feasible solution. In practical problems, it is possible that the set of constraints does not allow for a feasible solution (e.g., x >= 3, x <=2). Such a situation might result from a mistake in the problem statement or an error in data entry. Redundant equality constraints or nearly identical inequality constraints in the problem formulation may lead to a false indication that no feasible solution exists. Although the set of equalities may have a solution in theory, rounding errors inherent in computer computations may make the simultaneous satisfaction of these equalities (and sometimes inequalities) impossible.

LP solution concepts

- **Unbounded model**: If there are feasible solutions for which the objective function can achieve arbitrarily large values (if maximizing) or arbitrarily small values (if minimizing), the model is said to be unbounded. When all variables are restricted to be nonnegative and have finite simple upper bounds, this condition is impossible. If no bounds are specified for some variables, the model may have an unbounded solution. However, since most decisions must take into account limitations on resources and laws of nature, such a model is probably a poor representation of the real problem.

Simplex algorithm

- The simplex algorithm, developed by George Dantzig in 1947, solves LP problems by constructing a feasible solution at a vertex of the polytope and then walking along a path on the edges of the polytope to vertices with non-decreasing values of the objective function until an optimum is reached for sure. In many practical problems, "stalling" occurs: Many pivots are made with no increase in the objective function. In rare practical problems, the usual versions of the simplex algorithm may actually "cycle". To avoid cycles, researchers developed new pivoting rules.
- In practice, the simplex algorithm is quite efficient and can be guaranteed to find the global optimum if certain precautions against cycling are taken. The simplex algorithm has been proved to solve "random" problems efficiently, i.e. in a cubic number of steps, which is similar to its behavior on practical problems.
- However, the simplex algorithm has poor worst-case behavior: Klee and Minty constructed a family of linear programming problems for which the simplex method takes a number of steps exponential in the problem size. In fact, for some time it was not known whether the linear programming problem was solvable in polynomial time, i.e. of complexity class P.

Interior point algorithm

- In contrast to the simplex algorithm, which finds an optimal solution by traversing the edges between vertices on a polyhedral set, interior-point methods move through the interior of the feasible region.
- The ellipsoid algorithm (Khachiyan) is the first worst-case polynomial-time algorithm for linear programming. To solve a problem which has n variables and can be encoded in L input bits, this algorithm uses O(n^4 L) pseudo-arithmetic operations on numbers with O(L) digits. Khachiyan's algorithm and his long standing issue was resolved by Leonid Khachiyan in 1979 with the introduction of the ellipsoid method. The convergence analysis has (real-number) predecessors, notably the iterative methods developed by Naum Z. Shor and the approximation algorithms by Arkadi Nemirovski and D. Yudin.

LP Duality

- *Primal problem*: Maximize $c^T x$ subject to $Ax <= b$, $x >= 0$
- Corresponding *dual problem*: Minimize $b^T y$ subject to $A^T y >= c$, $y >= 0$
- **Weak duality theorem**: The objective function value of the dual solution is always greater than or equal to the objective function value of the primal at any feasible solution.
- **Strong duality theorem**: If the primal has an optimal solution, x*, then the dual also has an optimal solution y*, and $c^T x^* = b^T y^*$
- Fact: the dual of a dual linear program is the original primal linear program.
- Fact: Every feasible solution for a linear program gives a bound on the optimal value of the objective function of its dual.

https://math.stackexchange.com/questions/243706/what-are-the-advantages-of-dual-of-a-problem

- Understanding the dual problem can lead to specialized algorithms for some important classes of LP problems
  - E.g., Hungarian algorithm for assignment problem, Network Simplex method
- The dual can be helpful for *sensitivity analysis*
  - Modifying primal's constraints can make original primal optimal solution infeasible, but only changes objective function or adds new variable to dual, so original dual solution is still feasible (and close to new optimal solution)
- Sometimes finding initial feasible solution to dual is much easier than finding one for the primal.
  - $Ax>=b,x>=0$, for $b>=0$, dual $A^ty<=c,y>=0$, for $c>=0$. Origin feasible for dual.
- Dual variables give *shadow prices* for primal constraints
  - E.g., profit maximization problem with resource constraint i. The value y_i of corresponding dual variable in optimal solution tells that you get an increase of in maximum profit for each unit increase in the amount of resource i
- Sometimes dual is just easier to solve
  - Problem with many constraints and few variables can be converted into one with few constraints and many variables.

Cutting plane method for ILP

- Cutting plane methods for MILP work by solving a non-integer linear program, the *linear relaxation* of the given integer program. The theory of Linear Programming dictates that under mild assumptions (if the linear program has an optimal solution, and if the feasible region does not contain a line), one can always find an extreme point or a corner point that is optimal. The obtained optimum is tested for being an integer solution. If it is not, there is guaranteed to exist a linear inequality that separates the optimum from the convex hull of the true feasible set. Finding such an inequality is the separation problem, and such an inequality is a cut. A cut can be added to the relaxed linear program. Then, the current non-integer solution is no longer feasible to the relaxation. This process is repeated until an optimal integer solution is found.

Gomory cut (for ILP)

- Cutting planes were proposed by Ralph Gomory in the 1950s as a method for solving integer programming and mixed-integer programming problems. However most experts, including Gomory himself, considered them to be impractical due to numerical instability, as well as ineffective because many rounds of cuts were needed to make progress towards the solution. Things turned around when in the mid-1990s Gérard Cornuéjols and co-workers showed them to be very effective in combination with branch-and-bound (called branch-and-cut) and ways to overcome numerical instabilities. Nowadays, all commercial MILP solvers use Gomory cuts in one way or another. Gomory cuts are very efficiently generated from a simplex tableau, whereas many other types of cuts are either expensive or even NP-hard to separate. Among other general cuts for MILP, most notably lift-and-project dominates Gomory cuts.

Gomory cut algorithm

Let an integer programming problem be formulated (in Standard Form) as:

$$\begin{aligned}&\text{Maximize } c^T x\\&\text{Subject to } Ax = b,\\&\qquad x \geq 0,\ x_i \text{ all integers.}\end{aligned}$$

The method proceeds by first dropping the requirement that the x_i be integers and solving the associated linear programming problem to obtain a basic feasible solution. Geometrically, this solution will be a vertex of the convex polytope consisting of all feasible points. If this vertex is not an integer point then the method finds a hyperplane with the vertex on one side and all feasible integer points on the other. This is then added as an additional linear constraint to exclude the vertex found, creating a modified linear program. The new program is then solved and the process is repeated until an integer solution is found.

Using the simplex method to solve a linear program produces a set of equations of the form

$$x_i + \sum \bar{a}_{i,j} x_j = \bar{b}_i$$

where x_i is a basic variable and the x_j's are the nonbasic variables. Rewrite this equation so that the integer parts are on the left side and the fractional parts are on the right side:

$$x_i + \sum \lfloor \bar{a}_{i,j} \rfloor x_j - \lfloor \bar{b}_i \rfloor = \bar{b}_i - \lfloor \bar{b}_i \rfloor - \sum (\bar{a}_{i,j} - \lfloor \bar{a}_{i,j} \rfloor) x_j.$$

For any integer point in the feasible region the right side of this equation is less than 1 and the left side is an integer, therefore the common value must be less than or equal to 0. So the inequality

$$\bar{b}_i - \lfloor \bar{b}_i \rfloor - \sum (\bar{a}_{i,j} - \lfloor \bar{a}_{i,j} \rfloor) x_j \leq 0$$

must hold for any integer point in the feasible region. Furthermore, nonbasic variables are equal to 0s in any basic solution and if x_i is not an integer for the basic solution x,

$$\bar{b}_i - \lfloor \bar{b}_i \rfloor - \sum (\bar{a}_{i,j} - \lfloor \bar{a}_{i,j} \rfloor) x_j = \bar{b}_i - \lfloor \bar{b}_i \rfloor > 0.$$

So the inequality above excludes the basic feasible solution and thus is a cut with the desired properties. Introducing a new slack variable x_k for this inequality, a new constraint is added to the linear program, namely

$$x_k + \sum (\lfloor \bar{a}_{i,j} \rfloor - \bar{a}_{i,j}) x_j = \lfloor \bar{b}_i \rfloor - \bar{b}_i,\ x_k \geq 0,\ x_k \text{ an integer.}$$

Truth table for wumpus world

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 7.9 A truth table constructed for the knowledge base given in the text. KB is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

Satisfiability

- A **sentence** (in logic) is **satisfiable** if it is true in, or satisfied by, *some* model. For example, the knowledge base, (R1 AND R2 AND R3 AND R4 AND R5), is satisfiable because there are three models in which it is true.
- 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. Many problems in computer science (including the planning graph one, and integer programming) are really satisfiability problems.
- Many specialized "SAT-solving" algorithms. But it can also be formulated as an 0-1 ILP (or more generally a CSP).

Conjunctive Normal Form (CNF)

- (A OR B OR C) AND (D OR E) AND (NOT F) AND …
- Every propositional formula can be converted into an equivalent formula that is in CNF. This transformation is based on rules about logical equivalences: the double negative law, De Morgan's laws, and the distributive law.

- Since all logical formulae can be converted into an equivalent formula in conjunctive normal form, proofs are often based on the assumption that all formulae are CNF. However, in some cases this conversion to CNF can lead to an exponential explosion of the formula. For example, translating the following non-CNF formula into CNF produces a formula with 2^n clauses

CNF

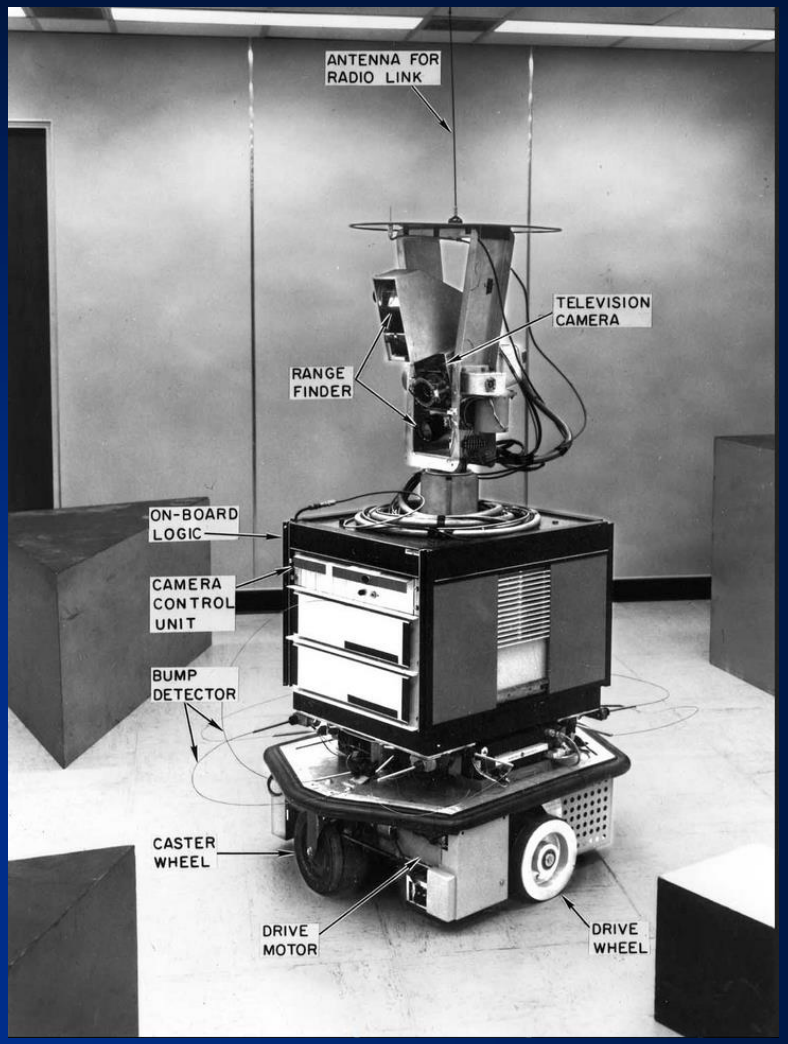
- (X1 AND Y1) OR (X2 AND Y2) OR … OR (Xn AND Yn)
- The generated formula is:
- (X1 OR X2 OR … OR Xn) AND (Y1 OR X2 OR … OR Xn) AND (X1 OR Y2 OR … OR Xn) AND (Y1 OR Y1 OR … OR Xn) AND (Y1 OR Y2 OR … OR Yn)
- Formula contains 2^n clauses, each clause contains either Xi or Yi for each i.
- An important set of problems in computational complexity involves finding assignments to the variables of a boolean formula expressed in Conjunctive Normal Form, such that the formula is true. The k-SAT problem is the problem of finding a satisfying assignment to a boolean formula expressed in CNF in which each disjunction contains at most k variables. 3-SAT is NP-complete (like any other k-SAT problem with k>2) while 2-SAT is known to have solutions in polynomial time.

Planning

- AI planning arose from investigations into state-space search, theorem proving, and control theory and from the practical needs of robotics, scheduling, and other domains.
- **Shakey the robot** was the first general-purpose mobile robot to be able to reason about its own actions. While other robots would have to be instructed on each individual step of completing a larger task, Shakey could analyze commands and break them down into basic chunks by itself.
- Due to its nature, the project combined research in robotics, computer vision, and natural language processing. Because of this, it was the first project that melded logical reasoning and physical action. Some of the most notable results of the project include the A* search algorithm, the Hough transform, and the visibility graph method.

Shakey

- https://www.youtube.com/watch?v=7bsEN8mwUB8



Fuzzy logic

- Fuzzy logic is a form of many-valued logic in which the truth values of variables may be any real number between 0 and 1. It is employed to handle the concept of partial truth, where the truth value may range between completely true and completely false. By contrast, in Boolean logic, the truth values of variables may only be the integer values 0 or 1. Furthermore, when linguistic variables are used, these degrees may be managed by specific (membership) functions. Fuzzy logic has been applied to many fields, from control theory to artificial intelligence.

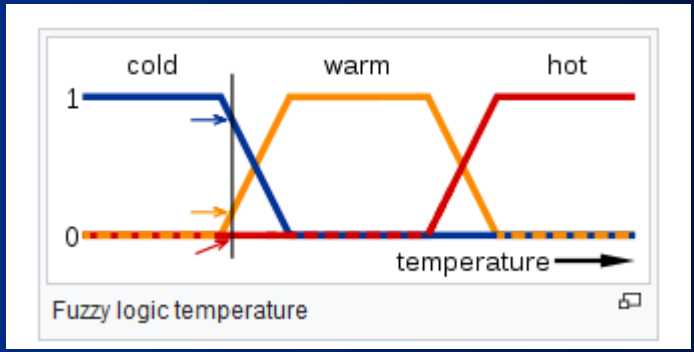
- Classical logic only permits conclusions which are either true or false. However, there are also propositions with variable answers, such as one might find when asking a group of people to identify a color. In such instances, the truth appears as the result of reasoning from inexact or partial knowledge in which the sampled answers are mapped on a spectrum.
- Humans and animals often operate using fuzzy evaluations in many everyday situations. In the case where someone is tossing an object into a container from a distance, the person does not compute exact values for the object weight, density, distance, direction, container height and width, and air resistance to determine the force and angle to toss the object. Instead he instinctively applies quick "fuzzy" estimates, based upon previous experience, to determine what output values of force, direction and vertical angle to use to make the toss.
- Both degrees of truth and probabilities range between 0 and 1 and hence may seem similar at first, but fuzzy logic uses degrees of truth as a mathematical model of vagueness, while probability is a mathematical model of ignorance.

Fuzzy logic

- A basic application might characterize various sub-ranges of a continuous variable. For instance, a temperature measurement for anti-lock brakes might have several separate membership functions defining particular temperature ranges needed to control the brakes properly. Each function maps the same temperature value to a truth value in the 0 to 1 range. These truth values can then be used to determine how the brakes should be controlled.
- 3-step process:
  1. Fuzzify all input values into fuzzy membership functions.
  2. Execute all applicable rules in the rulebase to compute the fuzzy output functions.
  3. De-fuzzify the fuzzy output functions to get "crisp" output values.

Fuzzification

- In this image, the meanings of the expressions cold, warm, and hot are represented by functions mapping a temperature scale. A point on that scale has three "truth values"—one for each of the three functions. The vertical line in the image represents a particular temperature that the three arrows (truth values) gauge. Since the red arrow points to zero, this temperature may be interpreted as "not hot". The orange arrow (pointing at 0.2) may describe it as "slightly warm" and the blue arrow (pointing at 0.8) "fairly cold".



Fuzzy logic temperature

Applications of fuzzy logic

- Many of the early successful applications of fuzzy logic were implemented in Japan. The first notable application was on the high-speed train in Sendai, in which fuzzy logic was able to improve the economy, comfort, and precision of the ride. It has also been used in recognition of hand written symbols in Sony pocket computers, flight aid for helicopters, controlling of subway systems in order to improve driving comfort, precision of halting, and power economy, improved fuel consumption for automobiles, single-button control for washing machines, automatic motor control for vacuum cleaners with recognition of surface condition and degree of soiling, and prediction systems for early recognition of earthquakes through the Institute of Seismology Bureau of Meteorology, Japan.
- FIU talk on climate change & national security used “fuzzy reasoning” https://www.cis.fiu.edu/lecture_series/climate-change-national-security/

Planning example: air cargo transport

- Three **actions**:
  - *Load, Unload, Fly*
- Two **predicates**:
  - *In(c,p)* means that cargo c is inside plane p
  - *At(x,a)* means that object x (either plane or cargo) is at airport a.
- **Initial state**
  - Conjunction (AND) of *ground atoms*. (Atoms that are not mentioned are false).
- **Goal**
  - Conjunction of literals
- **Preconditions** and **effects**
  - Must be specified for each action

Air cargo transport problem

$Init(At(C_1, SFO) \land At(C_2, JFK) \land At(P_1, SFO) \land At(P_2, JFK)$
$\land Cargo(C_1) \land Cargo(C_2) \land Plane(P_1) \land Plane(P_2)$
$\land Airport(JFK) \land Airport(SFO))$
$Goal(At(C_1, JFK) \land At(C_2, SFO))$
$Action(Load(c, p, a),$
PRECOND: $At(c, a) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $\neg At(c, a) \land In(c, p))$
$Action(Unload(c, p, a),$
PRECOND: $In(c, p) \land At(p, a) \land Cargo(c) \land Plane(p) \land Airport(a)$
EFFECT: $At(c, a) \land \neg In(c, p))$
$Action(Fly(p, from, to),$
PRECOND: $At(p, from) \land Plane(p) \land Airport(from) \land Airport(to)$
EFFECT: $\neg At(p, from) \land At(p, to))$

Figure 10.1 A PDDL description of an air cargo transportation planning problem.

Air cargo transport problem

- Note that some care must be taken to make sure the *At* predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic **PDDL** (Planning Domain Definition Language) does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be *At* anywhere when it is *In* a plane; the cargo only becomes *At* the new airport when it is unloaded. So *At* really means "available for use at a given location."
- PDDL based off STRIPS language.

STRIPS

- In artificial intelligence, STRIPS (Stanford Research Institute Problem Solver) is an automated planner developed by Richard Fikes and Nils Nilsson in 1971 at SRI International. The same name was later used to refer to the formal language of the inputs to this planner. This language is the base for most of the languages for expressing automated planning problem instances in use today.
- STRIPS instance is quadruple <P,O,I,G>
  - P is set of *conditions*
  - O is set of *operators* (i.e., actions). Each action specifies preconditions and postconditions .
  - I is initial state (set of conditions that are initially true).
  - G is *goal state* (set of conditions needed to be true/false to achieve goal).

Air cargo transport problem

- What is a solution for this problem?

Air cargo transport problem

- One solution (there may be others):

[Load(C1,P1,SFO), Fly(P1,SFO,JFK), Unload(C1,P1,JFK),
Load(C2,P2,JFK), Fly(P2,JFK,SFO), Unload(C2,P2,SFO)].

Air cargo transport problem

- What about “degenerate” actions like *Fly(P1,JFK,JFK)*?
- This should be a **no-op** (no operation), but it apparently has contradictory effects according to the definition (the effect would include At(P1,JFK) AND !At(P1,JFK)).
- It is common to ignore such problems and assume that the effects just cancel out. A perhaps better approach is to add inequality preconditions saying that the *from* and *to* airports must be different. We will see another similar example shortly.

Spare tire problem

- The goal is to have a good spare tire properly mounted onto the car's axle, where the initial state has a flat tire on the axle and a good spare tire in the trunk.
- Four actions:
  - Removing the spare tire from the trunk
  - Removing the flat tire from the axle
  - Putting the spare on the axle
  - Leaving the car unattended overnight
- Assume that the car is parked in a particularly bad neighborhood, so that the effect of leaving it overnight is that the tire disappear.

Spare tire problem

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
PRECOND: $At(obj, loc)$
EFFECT: $\neg At(obj, loc) \land At(obj, Ground))$
$Action(PutOn(t, Axle),$
PRECOND: $Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle) \land \neg At(Spare, Axle)$
EFFECT: $\neg At(t, Ground) \land At(t, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
$\land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))$

Figure 10.2 The simple spare tire problem.

Spare tire problem

- Solution?

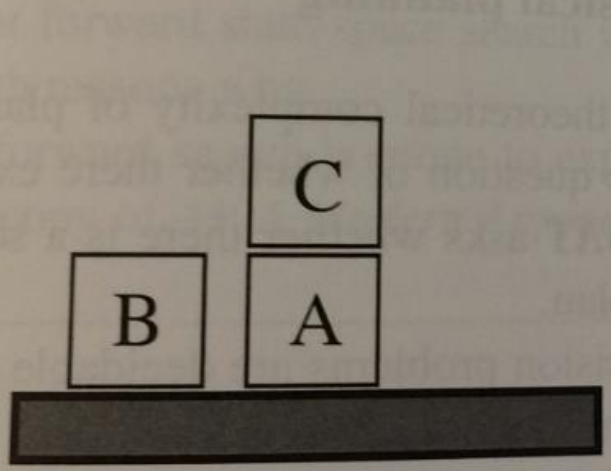
Spare tire problem

- [Remove(Flat, Axle), Remove(Spare, Trunk), PutOn(Spare, Axle)].

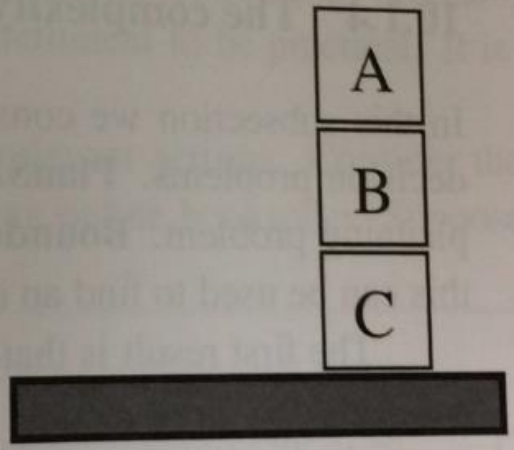
Blocks world

- One of the most famous planning domains is known as the **blocks world**. This domain consists of a set of cube-shaped blocks sitting on a table. The blocks can be stacked, but only one block can fit directly on top of another. A robot arm can pick up a block and move it to another position, either on the table or on top of another block. The arm can pick up only one block at a time, so it cannot pick up a block that has another one on it. The goal will always be to build one or more stacks of blocks, specified in terms of what blocks are on top of what other blocks. For example, a goal might be to get block A on B and block B on C.

Blocks world



Start State



Goal State

Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

Blocks world

- We use *On(b,x)* to indicate that block *b* is on *x*, where x is either another block or the table. The action for moving block b from the top of x to the top of y will be Move(b,x,y). One of the preconditions on moving b is that no other block be on it. In first-order logic, this would be !Exists x On(x,b), or alternatively, ForAll x ~On(x,b). Basic PDDL does not allow quantifiers, so instead we introduce a predicate *Clear*(x) that is true when nothing is on x.

Blocks world

$$
\begin{aligned}
&Init(On(A, Table) \wedge On(B, Table) \wedge On(C, A) \\
&\quad \wedge Block(A) \wedge Block(B) \wedge Block(C) \wedge Clear(B) \wedge Clear(C) \wedge Clear(Table)) \\
&Goal(On(A, B) \wedge On(B, C)) \\
&Action(Move(b, x, y), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Clear(y) \wedge Block(b) \wedge Block(y) \wedge \\
&\qquad (b \neq x) \wedge (b \neq y) \wedge (x \neq y), \\
&\quad \text{EFFECT: } On(b, y) \wedge Clear(x) \wedge \neg On(b, x) \wedge \neg Clear(y)) \\
&Action(MoveToTable(b, x), \\
&\quad \text{PRECOND: } On(b, x) \wedge Clear(b) \wedge Block(b) \wedge Block(x), \\
&\quad \text{EFFECT: } On(b, Table) \wedge Clear(x) \wedge \neg On(b, x))
\end{aligned}
$$

Blocks world

- Solution?

Blocks world

- [MoveToTable(C,A), Move(B,Table,C), Move(A,Table,B)]

Blocks world

- The action *Move* moves a block b from x to y if both b and y are clear. After the move is made, b is still clear but y is not. A first at the *Move* schema is
- Action(Move(b,x,y),
  - Precond: On(b,x) AND Clear(b) AND Clear(y)
  - Effect: On(b,y) AND Clear(X) AND ~On(b,x) AND ~Clear(y).

Blocks world

- Unfortunately, this does not maintain *Clear* properly when x or y is the table. When x is the Table, this action has the effect *Clear(Table)*, but the table should not become clear; and when *y=Table*, it has the precondition *Clear(Table)*, but the table does not have to be clear for us to move a block onto it. To fix this, we do two things. First we introduce another action to move a block b from x to the table:
- Action (MoveToTable(b,x),
  - Precond: On(b,x) AND Clear(b)
  - Effect: On(b,Table) AND Clear(x) AND ~On(b,x))

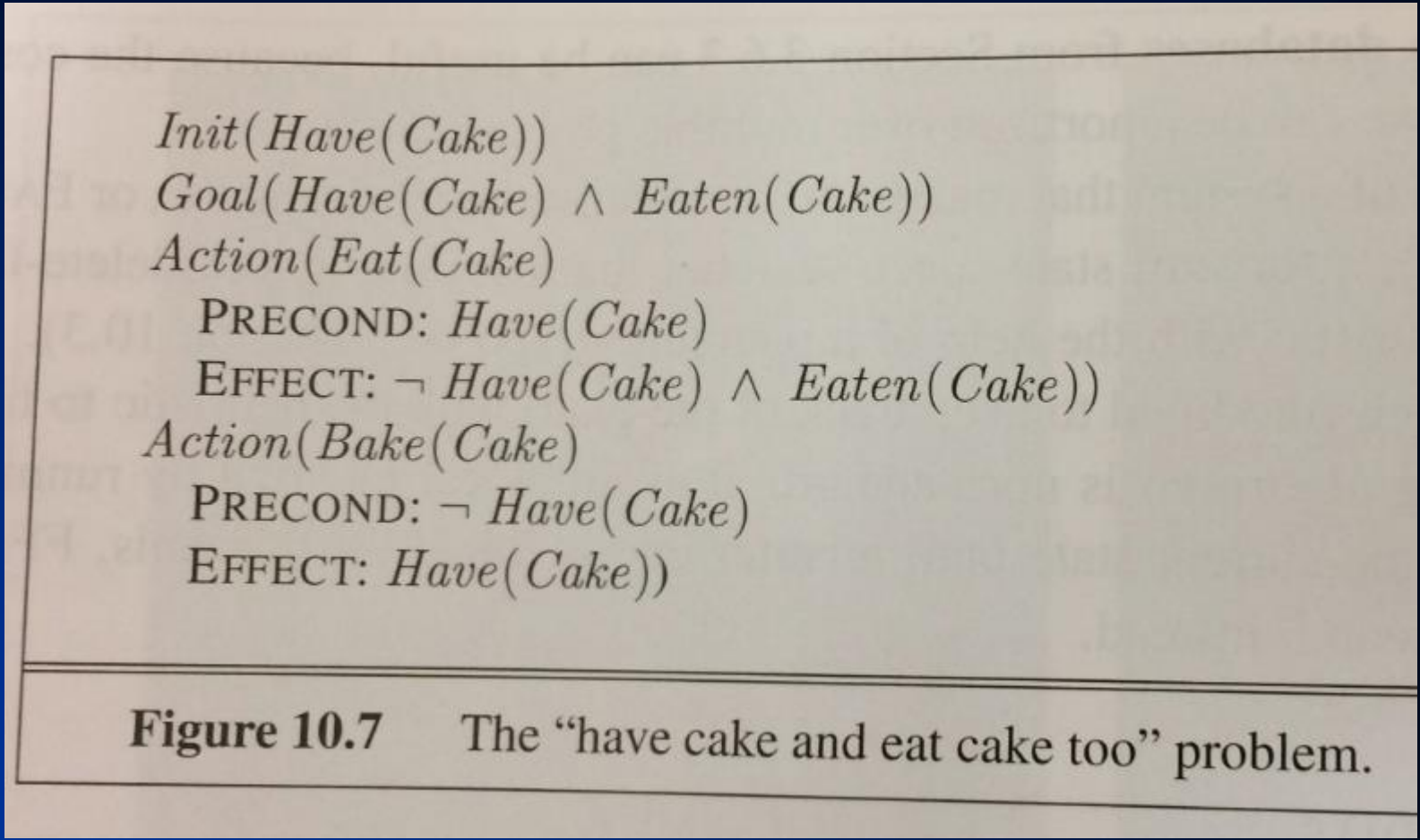
Blocks world

- Second, we take the interpretation of Clear(x) to be "there is a clear space on x to hold a block." Under this interpretation, Clear(Table) will always be true. The only problem is that nothing prevents the planner from using Move(b,x,Table) instead of MoveToTable(b,x), which leads to a larger than needed search space, though functionally is not problematic. We can fix this by introducing the predicate *Block* and add *Block(b) AND Block(y)* to the precondition of *Move*.

Planning in relation to other class modules

- We have seen that planning and search are very intertwined for robotics (e.g., Shakey implements A* search).
- Resemblance between Planning Domain Definition Language and First Order Logic.
- Planning graph can be represented as a **Satisfiability** problem in **Conjunctive-Normal Form** (conjunction (or AND) of clauses), which is an instance of constraint satisfaction.
- Certain AI planning models also solved by integer programming http://www.cs.umd.edu/~nau/papers/vossen1999use.pdf

Have cake and eat cake too



$Init(Have(Cake))$
$Goal(Have(Cake) \land Eaten(Cake))$
$Action(Eat(Cake)$
PRECOND: $Have(Cake)$
EFFECT: $\neg Have(Cake) \land Eaten(Cake))$
$Action(Bake(Cake)$
PRECOND: $\neg Have(Cake)$
EFFECT: $Have(Cake))$

Figure 10.7 The "have cake and eat cake too" problem.

Planning graph

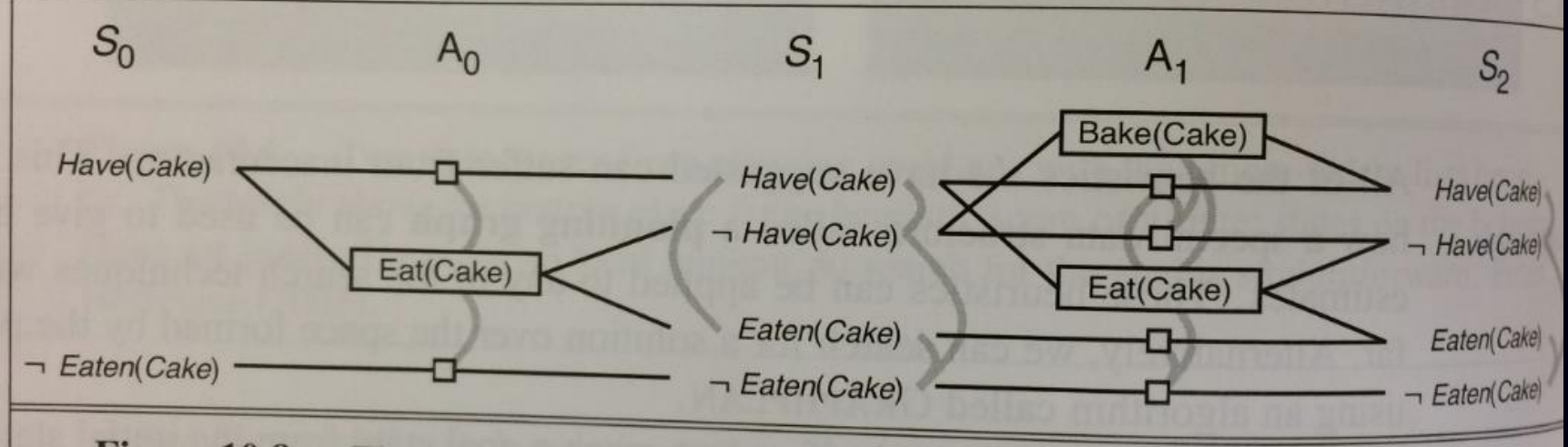


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

Planning graph

- A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their successors, and so on. If we indexed this tree appropriately, we could answer the planning question "can we reach state G from state S_0" immediately, by just looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is a polynomial-size approximation to this tree that can be constructed quickly. The planning graph can't answer definitively whether G is reachable from S_0, but it can *estimate* how many steps it takes to reach *G*. The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

Planning graph

- A planning graph is a directed graph organized into **levels**: first a level $S_{0,}$ for the initial state, consisting of nodes representing each fluent that holds in S_0; then a level A_0 consisting of nodes for each ground action that might be applicable in S_0; then alternating levels S_i followed by A_i; until we reach a termination condition.

Planning graphs

- Roughly speaking, S_i contains all the literals that *could* hold at time i, depending on the actions executed at preceding time steps. If it is possible that either P or !P could hold, then both will be represented in S_i. Also roughly speaking, A_i contains all the actions that *could* have their preconditions satisfied at time i. We say "roughly speaking" because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level S_j when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

Planning graphs

- The figure shows a simple planning problem “have cake and eat cake too” and its planning graph. Each action at level A_i is connected to its preconditions at S_i and its effects at S_{i+1}. So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it. This is represented by a **persistence action** (sometimes called a *no-op*). For every literal C, we add to the problem a persistence action with precondition C and effect C. Level A_0 shows one “real” action, *Eat(Cake)*, along with two persistence actions drawn as small square boxes.

- Level A_0 contains all the actions that *could* occur in state S_0, but just as important it records conflicts between actions that would prevent them from occurring together. The gray lines indicate **mutual exclusion** (or **mutex**) links. For example, *Eat(Cake)* is mutually exclusive with the persistence of either *Have(Cake)* or *!Eaten(Cake)*.

- Level S1 contains all the literals that could result from picking any subset of the actions in A_0, as well as mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, *Have(Cake)* and *Eaten(Cake)* are mutex: depending on the choice of actions in A_0, either, but not both, could be the result. In other words, S_1 represents a belief state: a set of possible states. The members of this set are all subsets of the literals such that there is no mutex link between any members of the subset.

Planning graphs

- We continue in this way, alternating between state level S_i and action level A_i until we reach a point where two consecutive levels are identical. At this point, we say that the graph has **leveled off**. The graph in the figure levels off at S_2.
- What we end up with is a structure where every A_i level contains all the actions that are applicable in S_i, along with constraints saying that two actions cannot both be executed at the same level. Every S_i level contains all the literals that could result from any possible choice of actions in A_{i-1}, along with constraints saying which pairs of literals are not possible. It is important to note that the process of constructing the planning graph does *not* require choosing among actions, which would entail combinatorial search. Instead, it just records the impossibility of certain choices using mutex links.

Planning graphs

- We now define mutex links for both actions and literals. A mutex relation holds between two *actions* at a given level if any of the following three conditions holds:
  - *Inconsistent effects*: one action negates an effect of the other. For example, *Eat(Cake)* and the persistence of *Have(Cake)* have inconsistent effects because they disagree on the effect *Have(Cake)*.
  - *Interference*: one of the effects of one action is the negation of a precondition of the other. For example *Eat(Cake)* interferes with the persistence of *Have(Cake)* by negating its precondition.
  - *Competing needs*: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, *Bake(Cake)* and *Eat(Cake)* are mutex because they compete on the value of the *Have(Cake)* precondition.

Planning graphs

- A mutex relation holds between two *literals* at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called *inconsistent support*. For example, *Have(Cake)* and *Eaten(Cake)* are mutex in S1 because the only way of achieving *Have(Cake)*, the persistence action, is mutex with the only way of achieving *Eaten(Cake)*, namely *Eat(Cake)*. In S2 the two literals are not mutex, because there are new ways of achieving them, such as *Bake(Cake)* and the persistence of *Eaten(Cake)*, that are not mutex.

Planning graphs

- A planning graph is polynomial in the size of the planning problem. For a planning problem with L literals and a actions, each Si has no more than L nodes and L^2 mutex links, and each Ai has no more than a+l nodes (including the no-ops), $(a+L)^2$ mutex links, and 2(aL+L) precondition and effect links. Thus, an entire graph with n levels has a size of $O(n(a+L)^2)$. The time to build the graph has the same complexity.

GRAPHPLAN algorithm

- The GRAPHPLAN algorithm repeatedly adds a level to a planning graph with EXPAND-GRAPH. Once all the goals show up as non-mutex in the graph, GRAPHPLAN calls EXTRACT-SOLUTION to search for a plan that solves the problem. If that fails, it expands another level and tries again, terminating with failure when there is no reason to go on.

GRAPHPLAN

```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
      if goals all non-mutex in S_t of graph then
          solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), nogoods)
          if solution ≠ failure then return solution
      if graph and nogoods have both leveled off then return failure
      graph ← EXPAND-GRAPH(graph, problem)
```

Figure 10.9 The GRAPHPLAN algorithm. GRAPHPLAN calls EXPAND-GRAPH to add a level until either a solution is found by EXTRACT-SOLUTION, or no solution is possible.

GRAPHPLAN

- Let us now trace the operation of GRAPHPLAN on the spare tire problem. The first line of GRAPHPLAN initializes the planning graph to a one-level (S_0) graph representing the initial state. The positive fluents (state variables) from the problem description's initial state are shown, as are the relevant negative fluents. Not shown are the unchanging positive literals (such as *Tire(Spare)*) and the irrelevant negative literals. The goal *At(Spare,Axle)* is not present in S_0, so we need not call EXTRACT-SOLUTION—we are certain that there is no solution yet. Instead, EXPAND-GRAPH adds into A_0 the three actions whose preconditions exist at level S_0 (i.e., all the actions except *PutOn(Spare, Axle)*, along with persistence actions for all the literals in S_0. The effects of the actions are added at level S_1. EXPAND-GRAPH then looks for mutex relations and adds them to the graph.

Spare tire problem

$Init(Tire(Flat) \land Tire(Spare) \land At(Flat, Axle) \land At(Spare, Trunk))$
$Goal(At(Spare, Axle))$
$Action(Remove(obj, loc),$
PRECOND: $At(obj, loc)$
EFFECT: $\neg At(obj, loc) \land At(obj, Ground))$
$Action(PutOn(t, Axle),$
PRECOND: $Tire(t) \land At(t, Ground) \land \neg At(Flat, Axle) \land \neg At(Spare, Axle)$
EFFECT: $\neg At(t, Ground) \land At(t, Axle))$
$Action(LeaveOvernight,$
PRECOND:
EFFECT: $\neg At(Spare, Ground) \land \neg At(Spare, Axle) \land \neg At(Spare, Trunk)$
$\land \neg At(Flat, Ground) \land \neg At(Flat, Axle) \land \neg At(Flat, Trunk))$

Figure 10.2 The simple spare tire problem.

GRAPHPLAN

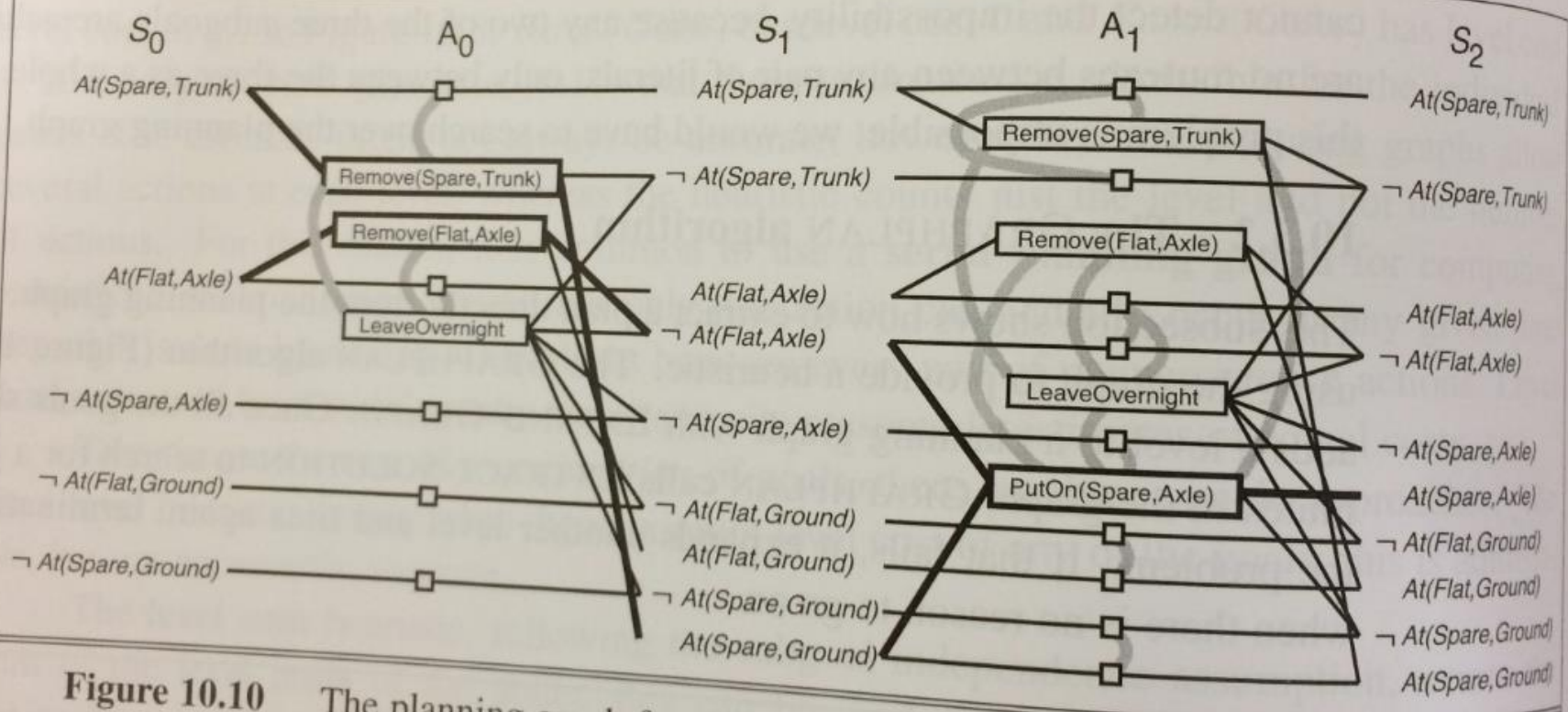


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

- *Interference: Remove(Flat,*

GRAPHPLAN

- *At(Spare, Axle)* is still not present in S1, so again we do not call EXTRACT-SOLUTION. We call EXPAND-GRAPH again, adding A_1 and S_1 and giving us the planning graph shown. Now that we have the full complement of actions, it is worthwhile to look at some of the examples of mutex relations and their causes:
  - *Inconsistent effects*: *Remove(Spare,Trunk)* is mutex with *LeaveOvernight* because one has the effect *At(Spare,Ground*) and the other has its negation.
  - *Interference*: *Remove(Flat, Axle)* is mutex with *LeaveOvernight* because one has the precondition *At(Flat,Axle)* and the other has its negation as an effect.
  - Competing needs: *PutOn(Spare, Axle)* is mutex with *Remove(Flat, Axle)* because one has At(Flat, Axle) as a precondition and other has its negation.
  - Inconsistent support: *At(Spare, Axle)* is mutex with *At(Flat, Axle)* in S_2 because the only way of achieving *At(Spare,Axle)* is by *PutOn(Spare, Axle)*, and that is mutex with the persistence action that is the only way of achieving *At(Flat,Axle)*. Thus, the mutex relations detect the immediate conflict that arises from trying to put two objects in the same place at the same time.

GRAPHPLAN

- This time, when we go back to the state of the loop, all the literals from the goal are present in S_2, and none of them is mutex with any other. That means that a solution might exist and EXTRACT-SOLUTION will try to find it. We can formulate EXTRACT-SOLUTION as a Boolean constraint satisfaction problem (CSP) where the variables are the actions at each level, the values for each variable are *in* or *out* of the plan, and the constraints are the mutexes and the need to satisfy each goal and precondition.

GRAPH-PLAN

- Alternatively, we can define EXTRACT-SOLUTION as a backward search problem, where each state in the search contains a pointer to a level in the planning graph and a set of unsatisfied goals.
  - Initial state is last level Sn with set of goals
  - Actions at Si are to select any "conflict-free" subset of actions in A_{i-1} whose effects cover the goals in the state. The resulting state has level S_{i-1} and has as its set of goals the preconditions for the selected set of actions. By "conflict free," we mean a set of actions such that no two of them are mutex and no two of their preconditions are mutex.
  - The goal is to reach a state at level S_0 such that all the goals are satisfied.
  - The cost of each action is 1.

GRAPHPLAN

- In the case where EXTRACT-SOLUTION fails to find a solution for a set of goals at a given level, we record the (level, goals) pair as a **no-good** (a similar idea is used for constraint learning for CSPs). Whenever EXTRACT-SOLUTION is called again with the same level and goals, we can find the recorded no-good and immediately return failure rather than searching again. We will see shortly that no-goods are also used in the termination test.

GRAPHPLAN

- We know that planning is PSPACE-complete (will elaborate next lecture) and that constructing the planning graph takes polynomial time, so it must be the case that solution extraction is intractable in the worst case. Therefore, we will need some heuristic guidance for choosing among actions during the backward search. One approach that works well in practice is a greedy algorithm based on the level cost of the literals. For any set of goals, we proceed in the following order:
  - Pick first the literal with the highest level cost
  - To achieve that literal, prefer actions with easier preconditions. That is, choose an action such that the sum (or maximum) of the level costs of its preconditions is smallest.

Planning summary

- Planning systems are problem-solving algorithms that operate on explicit propositional or relational representations of states and actions. These representations make possible the derivation of effective heuristics and the development of powerful and flexible algorithms for solving problems.
- PDDL, the Planning Domain Definition Language, describes the initial and goal states as conjunctions of literals, and actions in terms of their preconditions and effects.
- A **planning graph** can be constructed incrementally, starting from the initial state. Each layer contains a superset of all the literals or actions that could occur at that time step and encodes mutual exclusion (mutex) relations among literals or actions that cannot co-occur. Planning graphs yield useful heuristics for state-space and partial-order planners and can be used directly in the GRAPHPLAN algorithm.

Homework for next class

- Chapter 15 from Russel/Norvig
- HW3 out 10/31, due 11/14
- Next lecture: Describe planning complexity and GRAPHPLAN algorithm, start Probability module.