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/multiagent decision making (HW4 out)
- 11/16: Multiagent decision making/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 due Monday 12/4 at 2PM on Moodle.
- Final exam on 12/14

Announcements

- HW3 out 10/31 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
- HW4 likely out 11/14 due 11/28

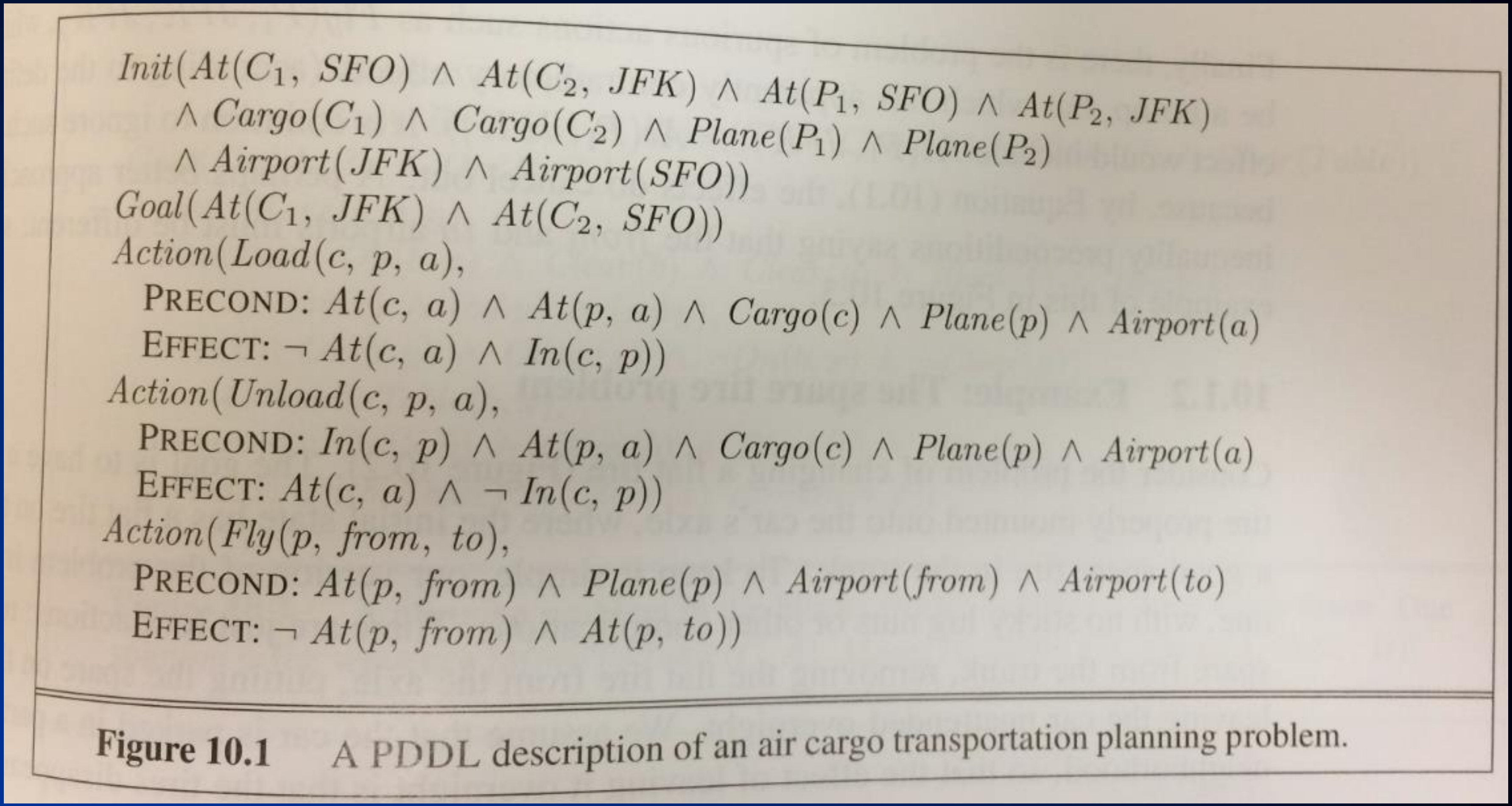
Class project

- 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
- This weekend I will post link for code and sample agent on Moodle

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)].

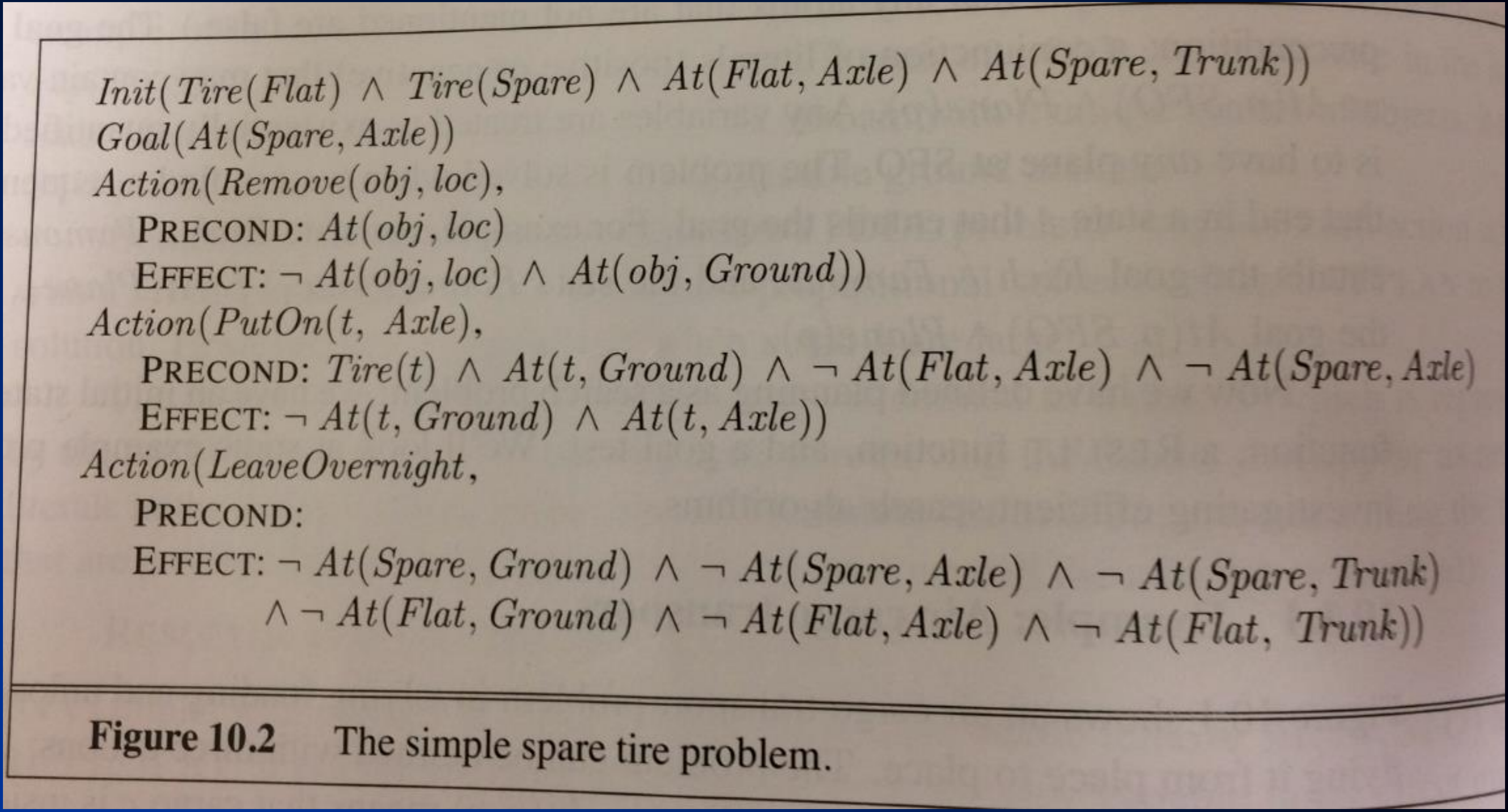
Air cargo transport problem

- 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

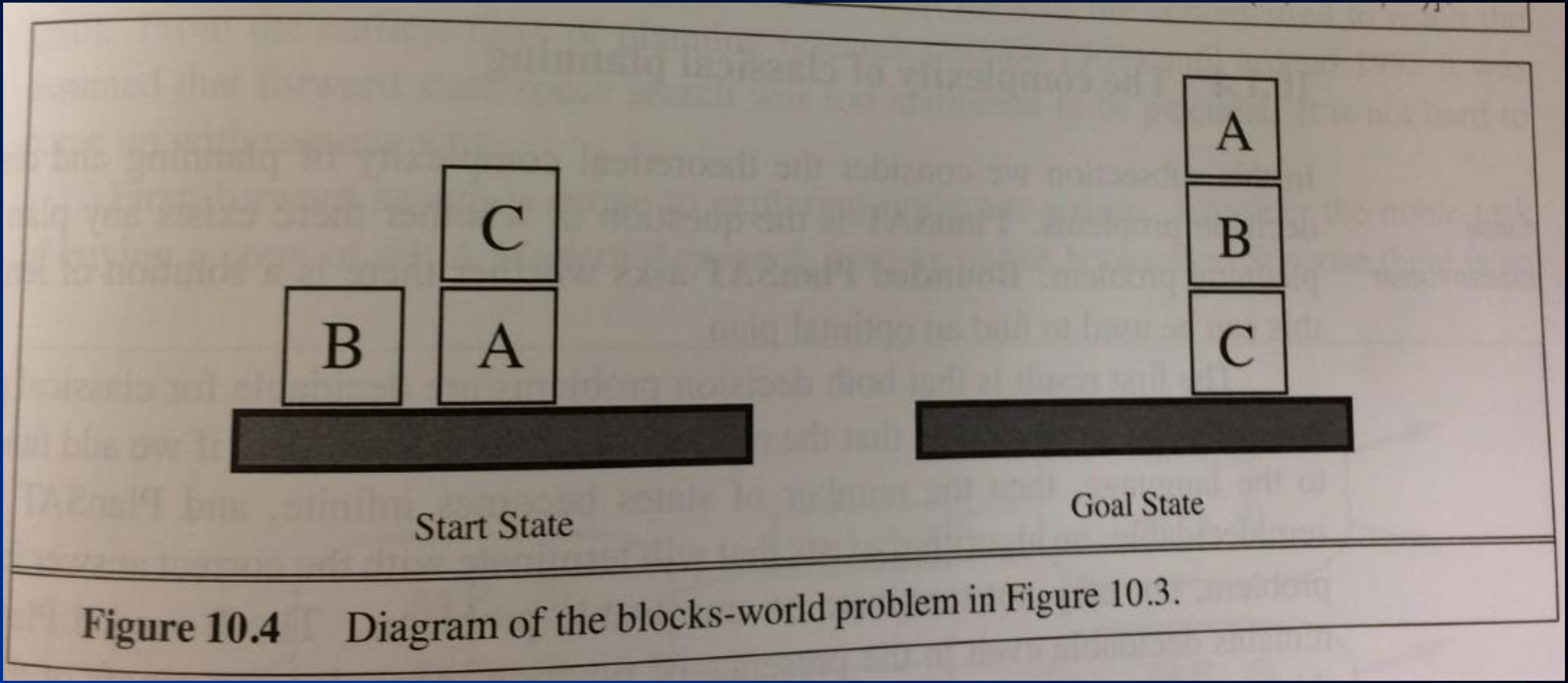
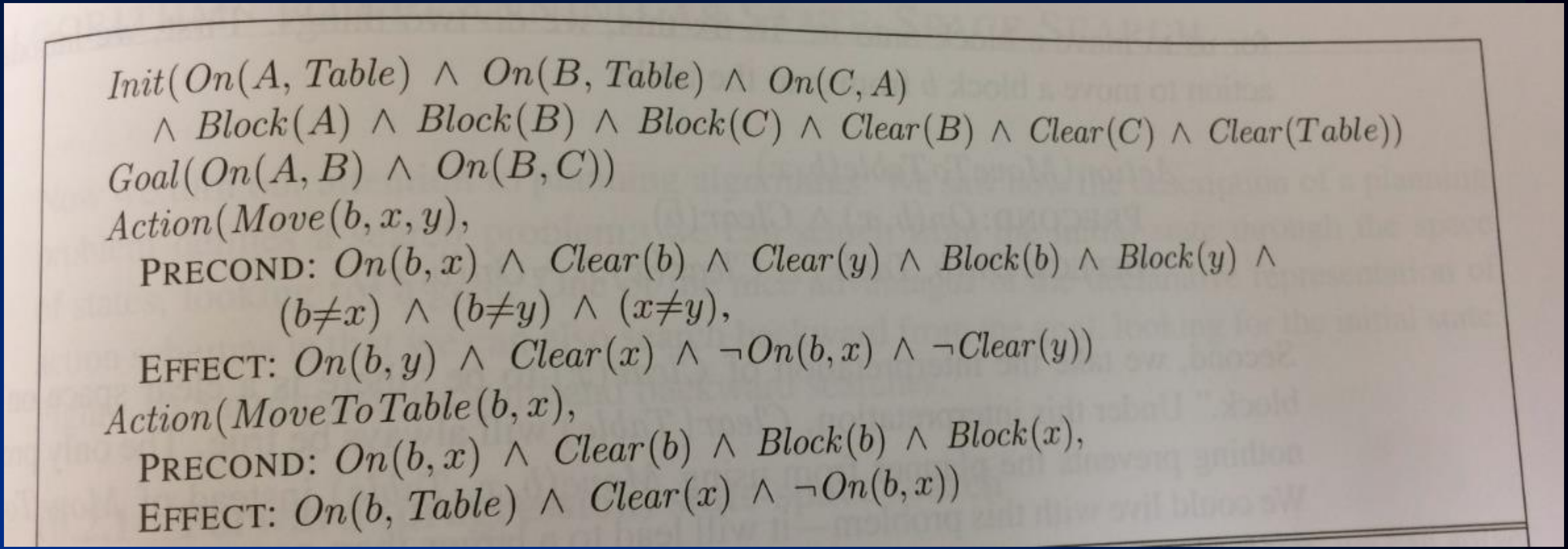


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

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



$Init(On(A, Table) \land On(B, Table) \land On(C, A)$
$\land\ Block(A) \land Block(B) \land Block(C) \land Clear(B) \land Clear(C) \land Clear(Table))$
$Goal(On(A, B) \land On(B, C))$
$Action(Move(b, x, y),$
PRECOND: $On(b, x) \land Clear(b) \land Clear(y) \land Block(b) \land Block(y) \land$
$(b \neq x) \land (b \neq y) \land (x \neq y),$
EFFECT: $On(b, y) \land Clear(x) \land \neg On(b, x) \land \neg Clear(y))$
$Action(MoveToTable(b, x),$
PRECOND: $On(b, x) \land Clear(b) \land Block(b) \land Block(x),$
EFFECT: $On(b, Table) \land Clear(x) \land \neg On(b, x))$

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))

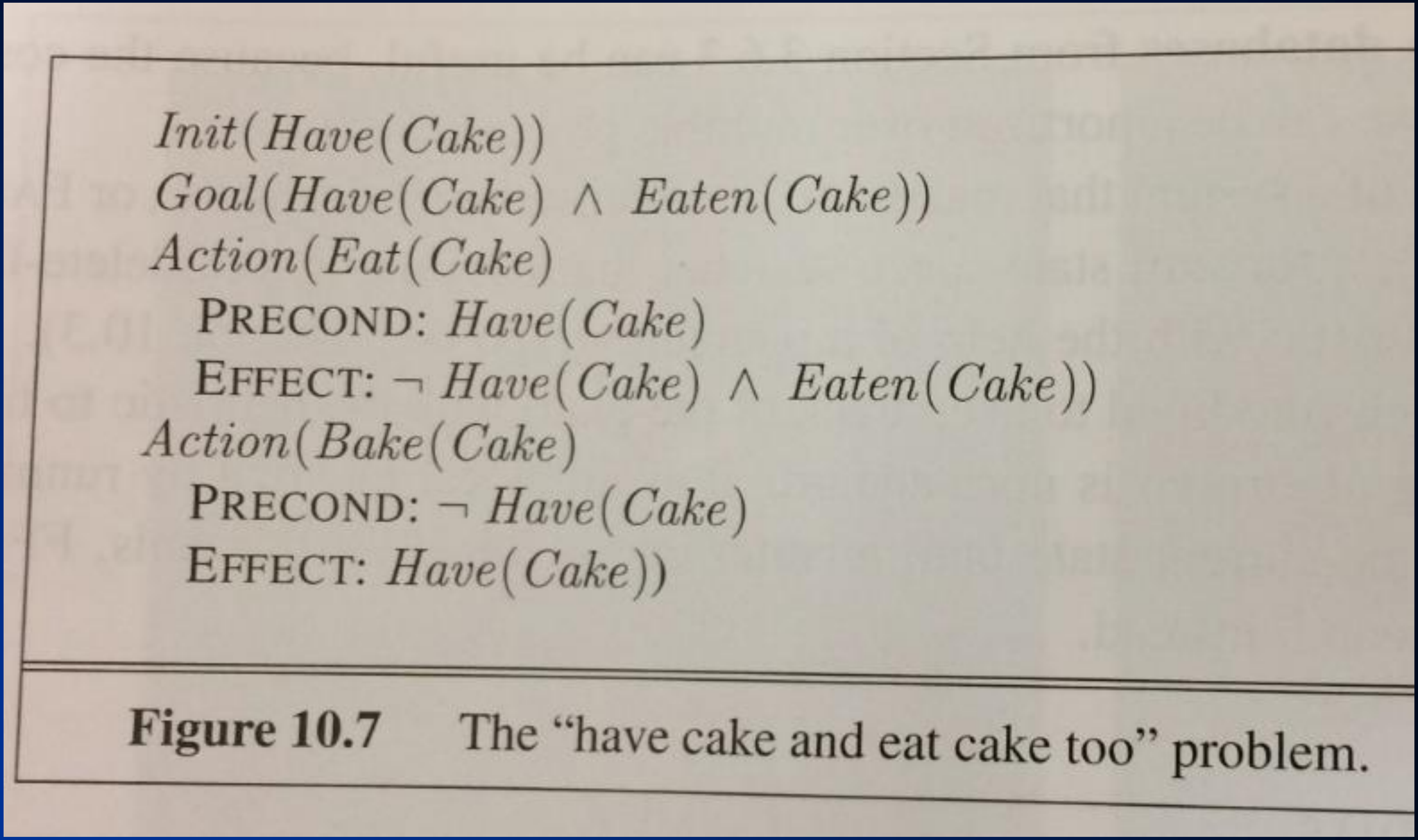
Blocks world

- 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

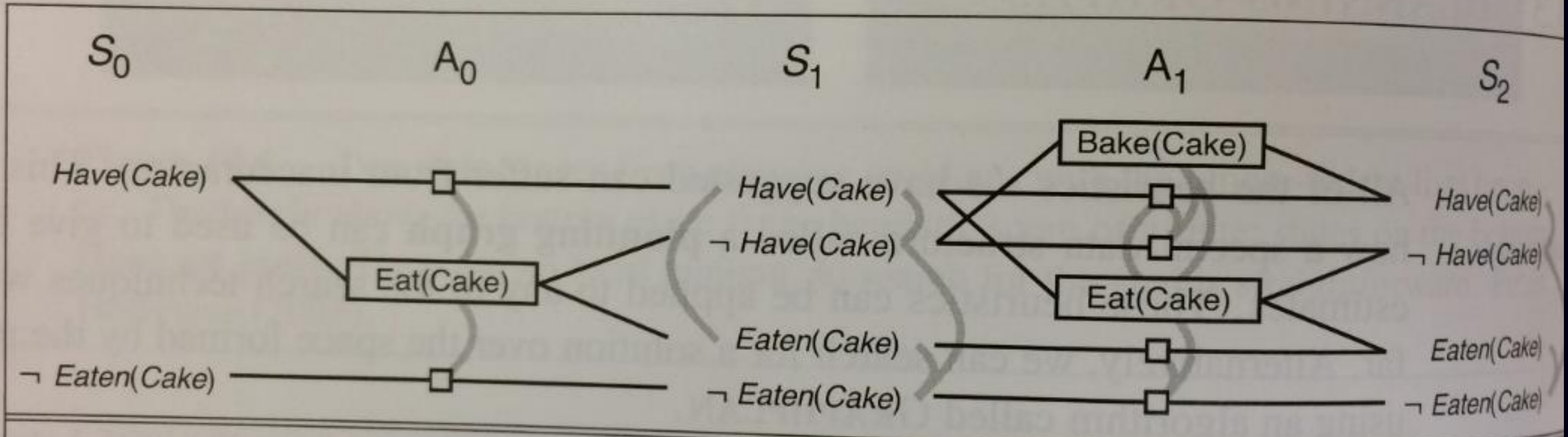


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because 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.

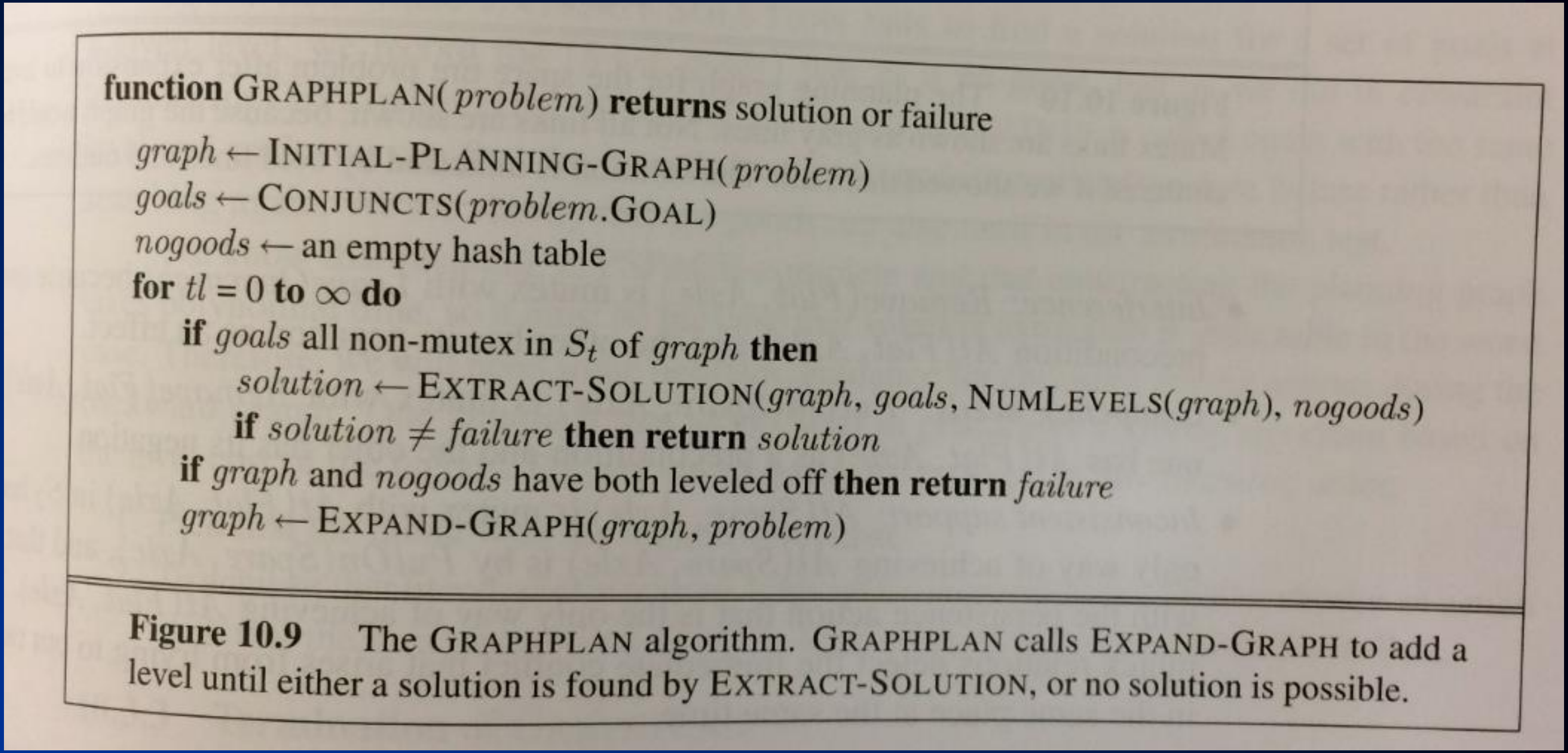
Planning graphs

- 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



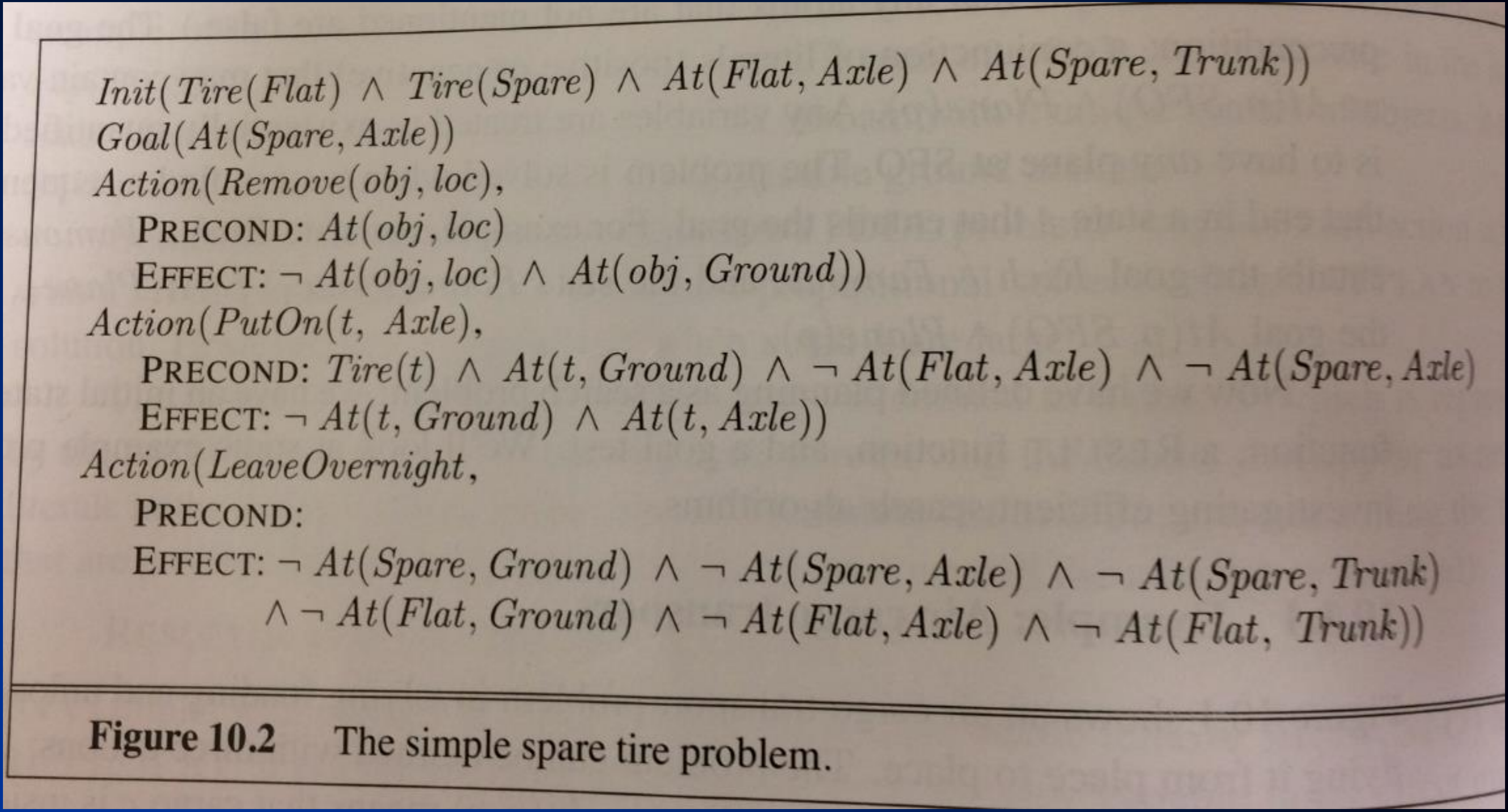
```
function GRAPHPLAN(problem) returns solution or failure
  graph ← INITIAL-PLANNING-GRAPH(problem)
  goals ← CONJUNCTS(problem.GOAL)
  nogoods ← an empty hash table
  for tl = 0 to ∞ do
      if goals all non-mutex in $S_t$ of graph then
          solution ← EXTRACT-SOLUTION(graph, goals, NUMLEVELS(graph), 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*) ∧ *Tire*(*Spare*) ∧ *At*(*Flat*, *Axle*) ∧ *At*(*Spare*, *Trunk*))
Goal(*At*(*Spare*, *Axle*))
Action(*Remove*(*obj*, *loc*),
  PRECOND: *At*(*obj*, *loc*)
  EFFECT: ¬ *At*(*obj*, *loc*) ∧ *At*(*obj*, *Ground*))
Action(*PutOn*(*t*, *Axle*),
  PRECOND: *Tire*(*t*) ∧ *At*(*t*, *Ground*) ∧ ¬ *At*(*Flat*, *Axle*) ∧ ¬ *At*(*Spare*, *Axle*)
  EFFECT: ¬ *At*(*t*, *Ground*) ∧ *At*(*t*, *Axle*))
Action(*LeaveOvernight*,
  PRECOND:
  EFFECT: ¬ *At*(*Spare*, *Ground*) ∧ ¬ *At*(*Spare*, *Axle*) ∧ ¬ *At*(*Spare*, *Trunk*)
    ∧ ¬ *At*(*Flat*, *Ground*) ∧ ¬ *At*(*Flat*, *Axle*) ∧ ¬ *At*(*Flat*, *Trunk*))

Figure 10.2 The simple spare tire problem.

GRAPHPLAN

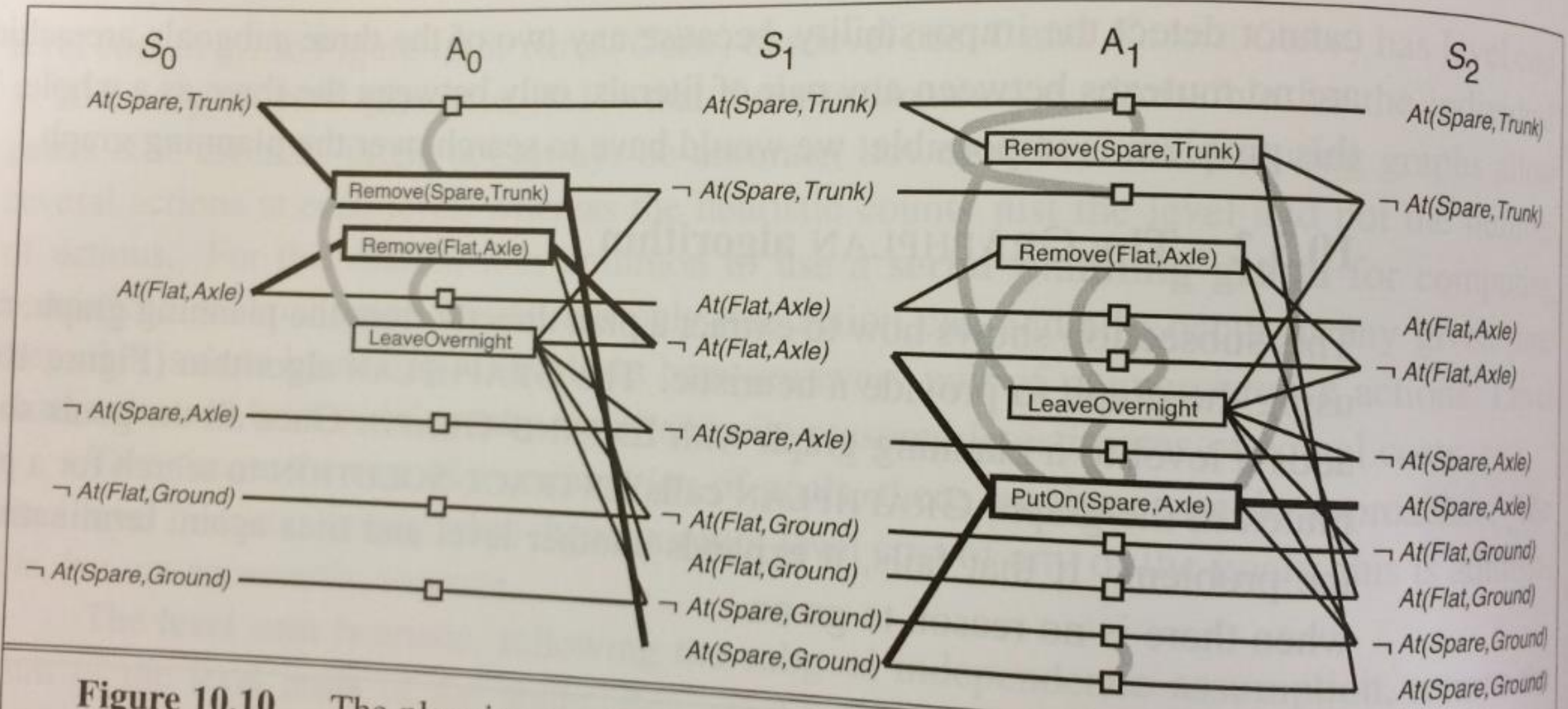


Figure 10.10 The planning graph for the spare tire problem after expansion to level S_2. Mutex links are shown as gray lines. Not all links are shown, because the graph would be too cluttered if we showed them all. The solution is indicated by bold lines and outlines.

• *Interference:* Remove(Fl

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.

GRAPHPLAN termination

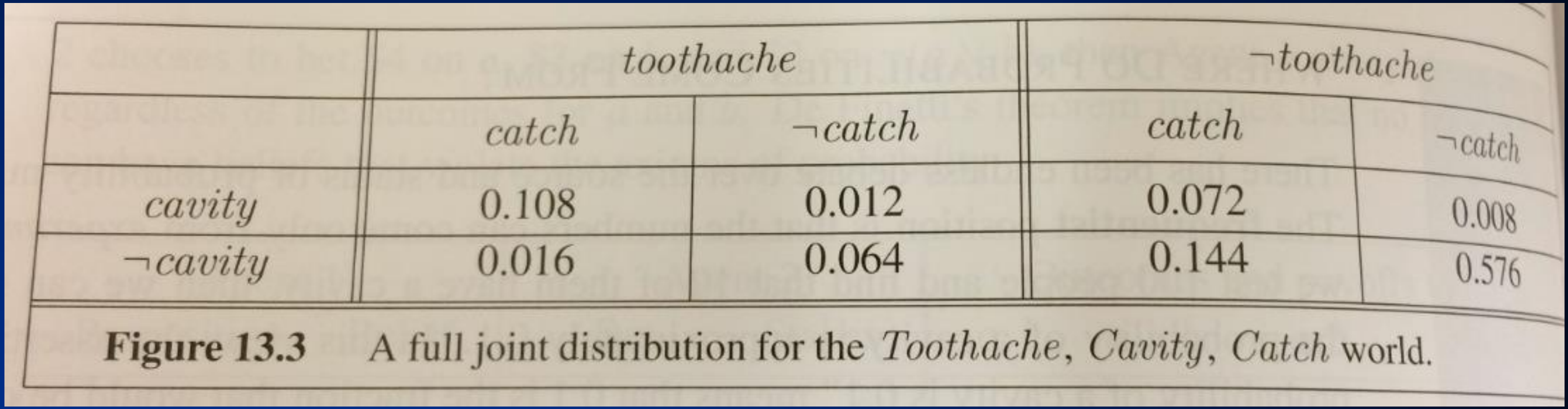
- How long do we have to keep expanding after the graph has leveled off? If the function EXTRACT-SOLUTION fails to find a solution, then there must have been at least one set of goals that were not achievable and were marked as a no-good. So it is possible that there might ne fewer no-goods in the next level, then we should continue. As soon as the graph itself and the no-goods have both leveled off, with no solution found, we can terminate with failure because there is no possibility of a subsequent change that could add a solution.

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.

Probability

- Consider a domain with three Boolean variables: *Toothache*, *Cavity*, *Catch* (the dentist's steel probe catches in my tooth).



	toothache		*¬toothache*	
	catch	*¬catch*	*catch*	*¬catch*
cavity	0.108	0.012	0.072	0.008
¬cavity	0.016	0.064	0.144	0.576

Figure 13.3 A full joint distribution for the *Toothache*, *Cavity*, *Catch* world.

Probability

- Notice that the probabilities in the **joint distribution** sum to 1, as required by the **axioms of probability**.
- Axioms of probability:
  1. 0 <= P(w) <= 1 for every possible world w
  2. Sum over all worlds w of P(w) = 1
- For example, if we roll two dice, there are 36 possible worlds: (1,1), (1,2), …, (6,6).
  - If each die is fair and rolls don’t interfere with each other, then each world has probability 1/36.
  - On the other hand, if the dice conspire to produce the same number, then the worlds (1,1), (2,2), (3,3), etc. might have higher probabilities, leaving the others with lower probabilities.

Probability

- Technique to calculate the probability of any proposition, simple or complex: identify those possible worlds in which the proposition is true and add up their probabilities. For example, there are six possible worlds in which *cavity OR toothache* holds:
  - P(*cavity OR toothache*) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064 = 0.28.
- One particularly common task is to extract the distribution over some subset of variables or a single variable. For example, adding the entries in the first row gives the **marginal probability** of *cavity*:
  - P(cavity) = 0.108 + 0.102 + 0.072 + 0.008 = 0.2.

Probability

- In general, for any sets of variables Y and Z, $P(Y) = \Sigma_{z \text{ in } Z} P(Y,z)$
- P(*Cavity*) = $\Sigma_{z \text{ in } \{\text{Catch, Toothache}\}}$ P(*Cavity*,z)
- Conditional probability:
  - P(a | b) = P(a AND b) / P(b) whenever P(b) > 0
  - P(*doubles* | Die1 = 5) = P(*doubles* AND Die1 = 5)/P(Die1 = 5)
- P(*cavity* | *toothache*) = P(*cavity AND toothache) / P(toothache)*

  = (0.108 + 0.012) / (.108 + 0.012 + 0.016 + 0.064) = 0.6.
- P(!*cavity* | *toothache*) = P(!*cavity AND toothache) / P(toothache)*

  = (0.016 + 0.064) / (.108 + 0.012 + 0.016 + 0.064) = 0.4.
- These two values sum to 1 as they should. This can be viewed as **normalization**.

Independence

- Let us expand the full joint distribution by adding a fourth variable, *Weather*. The full joint distribution then becomes P(Toothache, Catch, Cavity, Weather), which has 2 x 2 x 2 x 4 = 32 entries. It contains four "editions" of the table shown, one for each kind of weather.
- How do these editions relate to each other and to the original three-variable table? For example, P(toothache, catch, cavity, cloudy) vs. P(toothache, catch, cavity)?
- We can use the **product rule**:

  P(toothache, catch, cavity, cloudy)

  = P(cloudy | toothache, catch, cavity) * P(toothache, catch, cavity).

Probability

- Now, unless one is in the deity business, one should not imagine that one's dental problems influence the weather. And for indoor dentistry, at least, it seems safe to say that the weather does not influence the dental variables.
- Therefore, the following assertion seems reasonable:

  P(cloudy | toothache, catch, cavity) = P(cloudy).
- From this, we can deduce

  P(toothache, catch, cavity, cloudy) = P(cloudy)P(toothache, catch, cavity).
- A similar equation exists for every entry in P(toothache, catch, cavity, weather). In fact, we can write the general equation:

  P(toothache, catch, cavity, weather) = P(toothache, catch, cavity) P(weather)

Probability

- Thus, the 32-element table for four variables can be constructed from one 8-element table and one 4-element table. This decomposition is illustrated schematically in next slide. The property we used is called **independence** (also **marginal independence** and **absolute independence**). In particular, the weather is independent of one’s dental problems. Independence between propositions a and b can be written as:
  - $P(a|b) = P(a)$ or
  - $P(b|a) = P(b)$ or
  - $P(a \text{ AND } b) = P(a)P(b)$

Independence

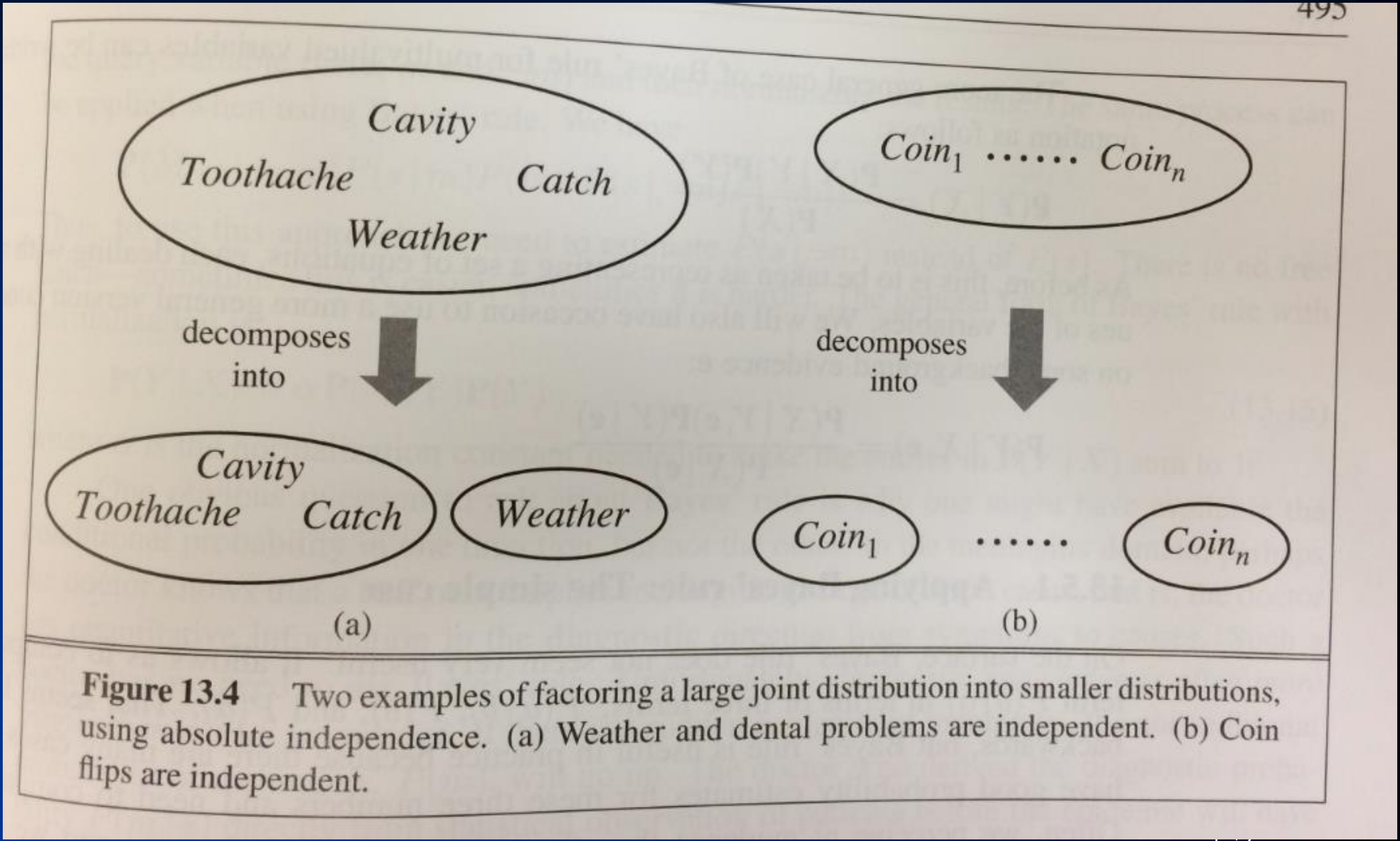


Figure 13.4 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

Independence

- Independence assertions are usually based on knowledge of the domain. As the toothache-weather example illustrates, they can dramatically reduce the amount of information necessary to specify the full joint distribution. If the complete set of variables can be divided into independent subsets, then the full joint distribution can be *factored* into separate joint distributions on those subsets. For example, the full joint distribution on the outcome of n independent coin flips, P(C1,…,Cn) has 2^n entries, but it can be represented as the product of n single-variable distributions P(Ci). In a more practical vein, the independence of dentistry and meteorology is a good thing, because otherwise the practice of dentistry might require intimate knowledge of meteorology, and vice versa.

Independence

- When they are available, then, independence assertions can help in reducing the size of the domain representation and the complexity of the inference problem. Unfortunately, clean separation of entire sets of variables by independence is quite rare. Whenever a connection, however indirect, exists between two variables, independence will fail to hold. Moreover, even independent subsets can be quite large—for example, dentistry might involve dozens of diseases and hundreds of symptoms, all of which are interrelated. To handle such problems, we need more subtle methods than the straightforward concept of independence.

Bayes' Rule

- Recall the **product rule**: P(a AND b) = P(a | b)P(b), or equivalently, P(a AND b) = P(b|a)P(a)
- Equating the two right-hand sides and dividing by P(a), we get
  - P(b|a) = P(a|b)P(b)/P(a)
- This equation is known as **Bayes' rule** (also Bayes' law or Bayes' theorem). This simple equation underlies most modern AI systems for probabilistic inference.

Bayes' rule

- On the surface, Bayes' rule does not seem very useful. It allows us to compute the single term P(b|a) in terms of three terms: P(a|b), P(b), and P(a). That seems like two steps backwards, but Bayes' rule is useful in practice because there are many cases where we do have good probability estimates for these three numbers and need to compute the fourth. Often, we perceive as evidence the *effect* of some unknown *cause* and we would like to determine that cause. In that case, Bayes' rule becomes
  - P(cause | effect) = P(effect | cause) P(cause) / P(effect)

Bayes' rule

- The conditional probability P(effect | cause) quantifies the relationship in the **causal** direction, whereas P(cause|effect) describes the **diagnostic** direction. In a task such as medical diagnosis, we often have conditional probabilities on causal relationships (that is, the doctor knows P(symptoms| disease) and want to derive a diagnosis, P(disease | symptoms). For example, a doctor knows that the disease meningitis causes the patient to have a stiff neck, say, 70% of the time. The doctor also knows some unconditional facts: the prior probability that a patient has meningitis is 1/50,000, and the prior probability that any patient has a stiff neck is 1%. Letting s be the proposition that the patient has a stiff neck and m be the proposition that the patient has meningitis, we have:

Bayes' rule

- P(s|m) = 0.7
- P(m) = 1/50000
- P(s) = 0.01
- P(m | s) = P(s|m)P(m)/P(s) = (0.7 * 1/5000)/0.01 = 0.0014
- Thus, we expect less than 1 in 700 patients with a stiff neck to have meningitis. Notice that even though a stiff neck is quite strongly indicated by meningitis (with probability 0.7), the probability of meningitis in the patient remains small. This is because the prior probability of stiff necks is much higher than that of meningitis.

Probability

- What is probability that sum of first two die rolls is >= 9 given that the first roll is 5?

Bayesian networks

- A Bayesian network is a directed graph in which each node is annotated with quantitative probability information. The full specification is:
  1. Each node corresponds to a random variable, which may be discrete or continuous
  2. A set of directed links or arrows connects pairs of nodes. If there is an arrow form node X to node Y, X is said to be a *parent* of Y. The graph has no directed cycles (and hence is a directed acyclic graph), or DAG.
  3. Each node Xi has a conditional probability distribution P(Xi|Parents(Xi)) that quantifies the effect of the parents on the node.

Bayesian networks

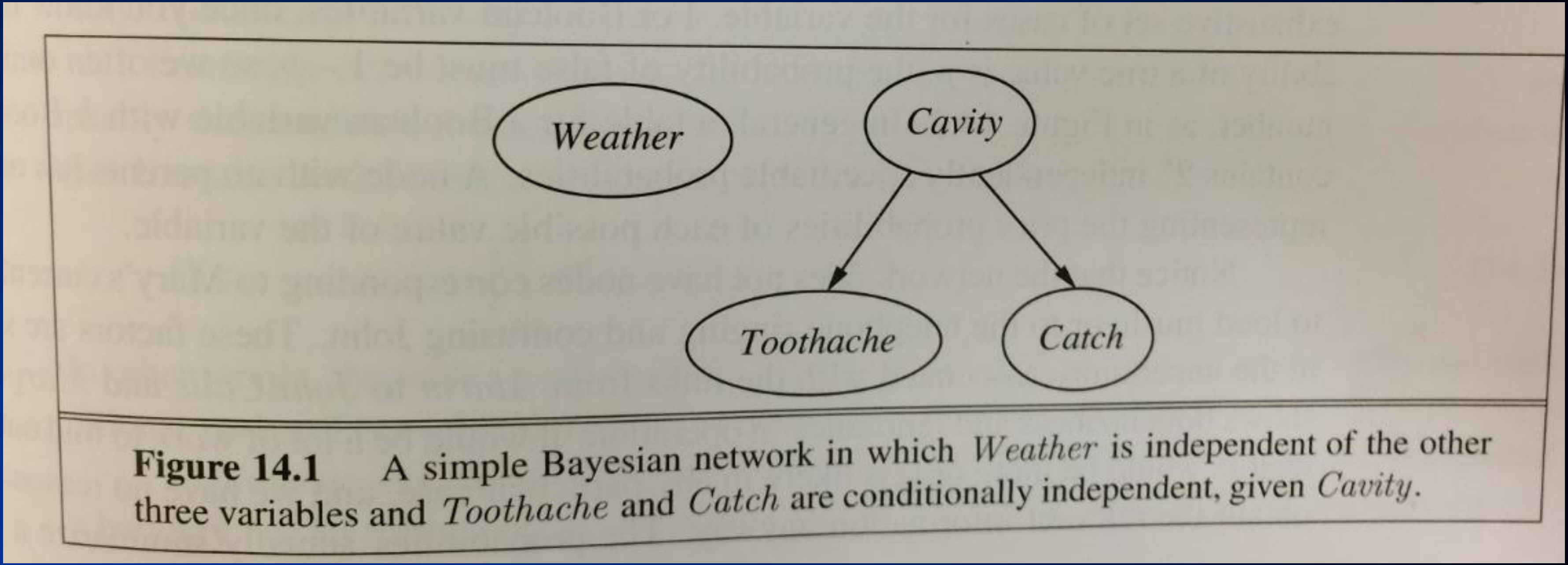
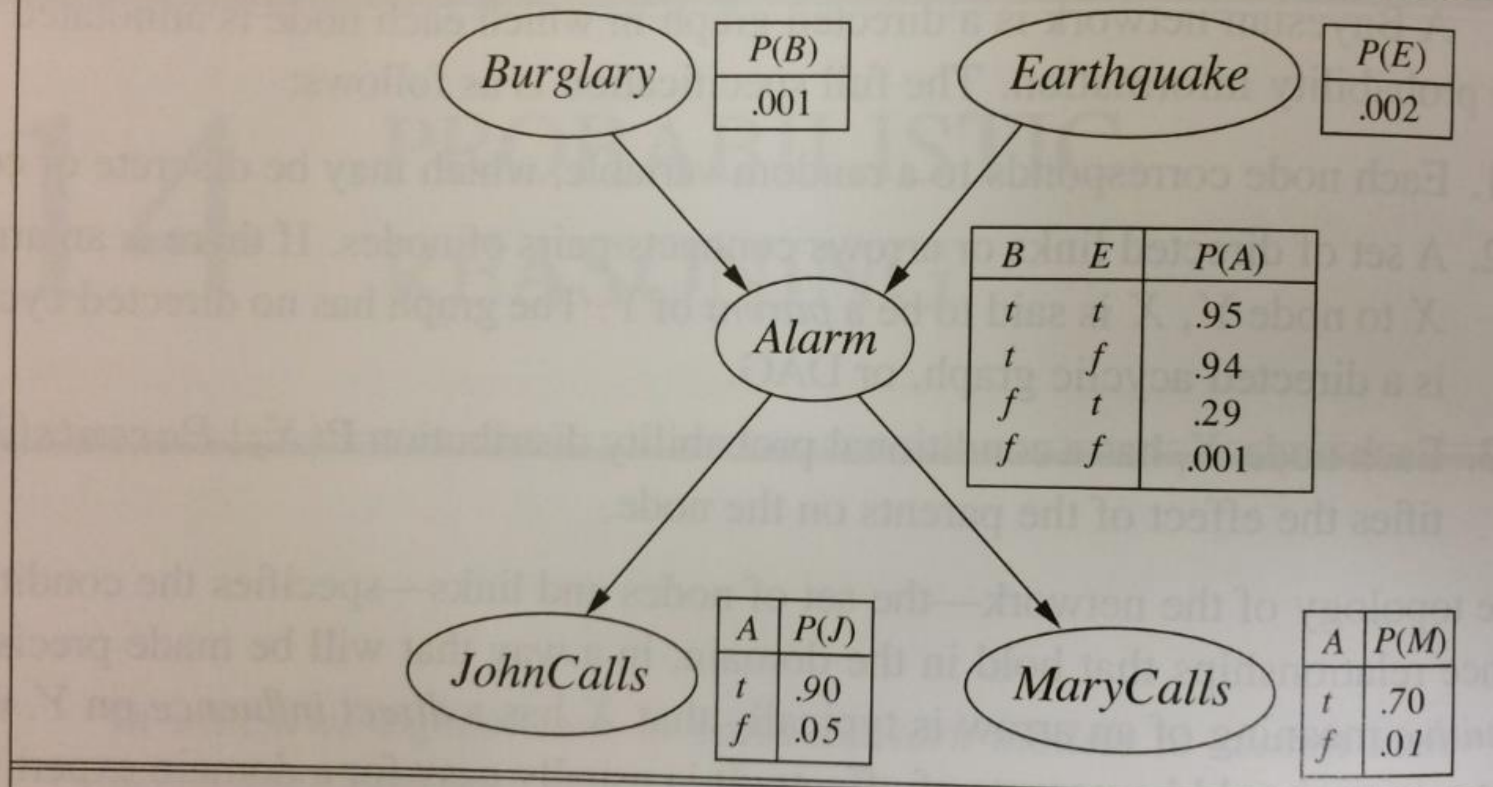


Figure 14.1 A simple Bayesian network in which *Weather* is independent of the other three variables and *Toothache* and *Catch* are conditionally independent, given *Cavity*.

Bayesian network



P(B)
.001

P(E)
.002

B	E	P(A)
t	t	.95
t	f	.94
f	t	.29
f	f	.001

A	P(J)
t	.90
f	.05

A	P(M)
t	.70
f	.01

Figure 14.2 A typical Bayesian network, showing both the topology and the conditional probability tables (CPTs). In the CPTs, the letters B, E, A, J, and M stand for *Burglary*, *Earthquake*, *Alarm*, *JohnCalls*, and *MaryCalls*, respectively.

Homework for next class

- Chapter 16 from Russel/Norvig
- HW3 out 10/31, due 11/14
- Next lecture: Finish up probability, describe extensions for Markov decision processes and multiagent decision making (game theory)