CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

Schedule

- 10/12: Wrap-up logic (logical inference), start optimization (integer, linear optimization)
- 10/17: Continue optimization (integer, linear optimization)
- 10/19: Wrap up optimization (nonlinear optimization), go over homework 1, midterm review
- 10/24: Midterm
- 10/26: Go over exam, start planning module
- Next week: HW2 back, HW3 out
- Class project due 12/7
- Final exam on 12/14
- Class withdrawal deadline is 11/6

Integer programming

- Special case of a CSP where domain set for each variable is a set of integers
  - Often it is finite {0,1,2,…,n} but could be infinite, {0,1,2,3,…..}
  - Often it is just binary {0,1}
- Constraints are all LINEAR functions of the variables
  - E.g., 4X1 + 3X2 <= 9
  - -2.5X1 + 2X2 – 19X3 <= 22
  - Cannot raise variables to powers or multiply variables together

Objective functions

- In most CSP examples we saw, the goal was just to find a single assignment of values to variables that satisfied all the constraints, and it did not matter which solution was found. We also considered the more general setting where we have “preference constraints” which are encoded as costs on individual variable assignments, leading to an overall objective function that want would like minimize, subject to all of the constraints being adhered to.

CSP variations

- The constraints we have described so far have all been absolute constraints, violation of which rules out a potential solution. Many real-world CSPs include **preference constraints** indicating which solutions are preferred. For example, in a university class-scheduling problem there are absolute constraints that no professor can teach two classes at the same time. But we also may allow preference constraints: Prof. R might prefer teaching in the morning, whereas Prof. N prefers teaching in the afternoon. A schedule that has Prof. R teaching at 2 p.m. would still be an allowable solution (unless Prof. R happens to be the department chair) but would not be an optimal one.

CSP variations

- Preference constraints can often be encoded as costs on individual variable assignments—for example, assigning an afternoon slot for Prof. R costs 2 points against the overall objective function, whereas a morning slot costs 1. With this formulation, CSPs with preferences can be solved with optimization search methods, either path-based or local. We call such a problem a **constraint optimization problem**, or COP. Linear/integer/nonlinear programming problems do this kind of optimization.

Integer programming

- Special case of a CSP where domain set for each (or some) variable is a set of integers
  - Often it is finite {0,1,2,…,n} but could be infinite, {0,1,2,3,…..}
  - Often it is just binary {0,1}
  - Some variables do not have integer restrictions and can be any real number
- Constraints are all LINEAR functions of the variables
  - E.g., 4X1 + 3X2 <= 9
  - -2.5X1 + 2X2 – 19X3 <= 22
  - Cannot raise variables to powers or multiply variables
- Objective function of the variables to optimize

Integer linear programming

- Often the constraints and the objective are both LINEAR functions of the variables, and we referring to integer programming (IP) as integer linear programming in this case (ILP). One could also consider other forms for the constraints and objective (e.g., quadratic program, quadratically-constrained program, conic program). Specialized algorithms exist for these as well, though more attention has been given to the linear case and typically those algorithms are much more effective in practice.

Manufacturing site selection

- A manufacturer is planning to construct new buildings at four local sites designated 1, 2, 3, and 4. At each site, there are three possible building designs labeled A, B, and C. There is also the option of not using a site. The problem is to select the optimal combination of building sites and building designs. Preliminary studies have determined the required investment and net annual income for each of the 12 options. This information is shown in Table 7.1 with A1, for example, denoting design A at site 1. The company has an investment budget of $100 million ($100M). The goal is to maximize total annual income without exceeding the investment budget. As the optimization analyst, you are given the job of finding the optimal plan.

Manufacturing site selection

- It is an obvious requirement here that only whole buildings may be built and only whole designs may be selected. To begin creating a model, variables must be defined to represent each decision. Let I = {A,B,C} be the set of design options, and let J = {1,2,3,4} be the set of site options.
- Let yij = 1 if design i is used at site j, and 0 otherwise
- Also, denote by pij the annual net income and by aij the investment required for the design/site combination i,j. As a first try, you propose the following model for finding the maximum of annual income:

Manufacturing site selection

- Maximize z = sum_i sum_j pij yij
- Subject to:
  - sum_i sum_j aij yij <= 100
  - yij in {0,1} for all i in I and j in J

Manufacturing site selection

- Solving the model with an appropriate algorithm for the parameter values given in the table, the optimal solution is:
  - yA1=yA3=yB3=yB4=yC1=1, with all other values of yij equal to zero and z = 40. Of the available budget, $99M is used.

Table 7.1 Data for Site Selection Example

Option	A1	A2	A3	A4	B1	B2	B3	B4	C1	C2	C3	C4
Net income ($M)	6	7	9	11	12	15	5	8	12	16	19	20
Investment ($M)	13	20	24	30	39	45	12	20	30	44	48	55

Manufacturing site selection

- Your supervisor reviews the solution and questions your basic reasoning. You seem to have omitted some of the logic of the problem, because two designs are built on the same site—that is, A1 and C1, and also A3 and B3, are all in the solution. In addition, your supervisor now realizes that you were not alerted to several other logical restrictions imposed by the owners and architects—i.e., site 2 must have a building, design A can be used at sites 1, 2, and 3 only if it is also selected for site 4, and at most two of the designs may be included in the plans.
- Your solution violates all of these restrictions and must be discarded. The following additional constraints are needed to guarantee a feasible solution:

Manufacturing site selection

- Site 2 must have a building: sum_i yi2 = 1
- There can be at most one building at each of the other sites: sum_i yij <= 1 for j = 1,3,4
- Design A can be used at sites 1, 2, and 3 only if it is also selected for site 4: yA1 + yA2 + yA3 <= 3yA4.
- To formulate the constraints associated with design selection, three new binary variables are introduced.
  - Let wi = 1 if design i is used, 0 otherwise, for i = A,B,C
  - At most two designs may be used: wA + wB + wC <= 2
  - Finally, the yij and wi variables must be tied together: sum_j yij <= 4wi for i = A, B, C

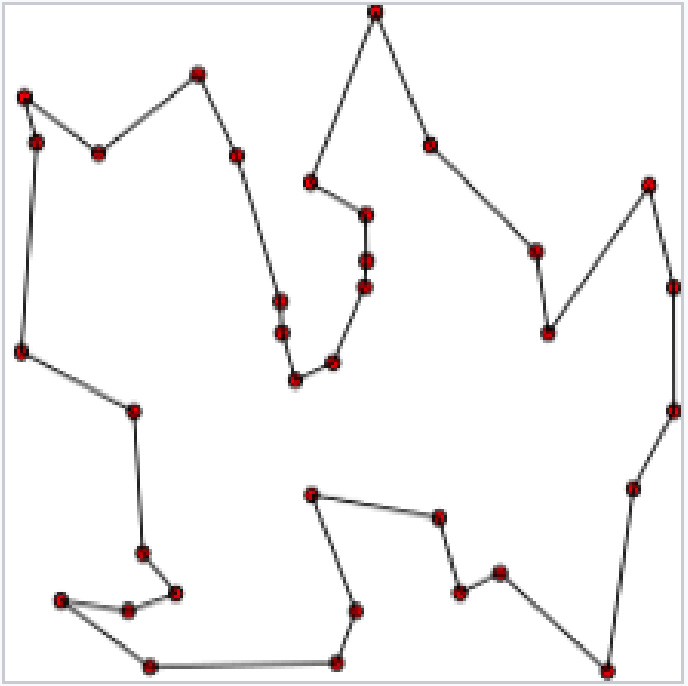
Manufacturing site selection

- The new model has 15 variables and 10 constraints not including the integrality requirement. Solving, you find that the optimal solution is yA1=yA4=yB2=yB3=wA=wB=1 with all other variables equal to zero and z = 37. All the budget is spent, but the profit has decreased.

Traveling salesman problem

- The **travelling salesman problem** (**TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?“
- The problem was first formulated in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact algorithms are known, so that some instances with tens of thousands of cities can be solved completely and even problems with millions of cities can be approximated within a small fraction of 1%.

Traveling salesman problem



Solution of a travelling salesman problem: the black line shows the shortest possible loop that connects every red dot

Traveling salesman problem

- The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources. In many applications, additional constraints such as limited resources or time windows may be imposed.

Traveling salesman problem

TSP can be formulated as an integer linear program.[10][11][12] Label the cities with the numbers 1, ..., *n* and define:

$$x_{ij} = \begin{cases} 1 & \text{the path goes from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}$$

For *i* = 1, ..., *n*, let u_i be a dummy variable, and finally take c_{ij} to be the distance from city *i* to city *j*. Then TSP can be written as the following integer linear programming problem:

$$\begin{aligned} &\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} c_{ij} x_{ij}: \\ &0 \leq x_{ij} \leq 1 && i, j = 1, \ldots, n; \\ &u_i \in \mathbf{Z} && i = 1, \ldots, n; \\ &\sum_{i=1, i \neq j}^{n} x_{ij} = 1 && j = 1, \ldots, n; \\ &\sum_{j=1, j \neq i}^{n} x_{ij} = 1 && i = 1, \ldots, n; \\ &u_i - u_j + n x_{ij} \leq n - 1 && 2 \leq i \neq j \leq n. \end{aligned}$$

The first set of equalities requires that each city be arrived at from exactly one other city, and the second set of equalities requires that from each city there is a departure to exactly one other city. The last constraints enforce that there is only a single tour covering all cities, and not two or more disjointed tours that only collectively cover all cities. To prove this, it is shown below (1) that every feasible solution contains only one closed sequence of cities, and (2) that for every single tour covering all cities, there are values for the dummy variables u_i that satisfy the constraints.

To prove that every feasible solution contains only one closed sequence of cities, it suffices to show that every subtour in a feasible solution passes through city 1 (noting that the equalities ensure there can only be one such tour). For if we sum all the inequalities corresponding to $x_{ij} = 1$ for any subtour of *k* steps not passing through city 1, we obtain:

$$nk \leq (n-1)k,$$

which is a contradiction.

It now must be shown that for every single tour covering all cities, there are values for the dummy variables u_i that satisfy the constraints.

Without loss of generality, define the tour as originating (and ending) at city 1. Choose $u_i = t$ if city *i* is visited in step *t* (*i*, *t* = 1, 2, ..., n). Then

$$u_i - u_j \leq n - 1,$$

since u_i can be no greater than *n* and u_j can be no less than 1; hence the constraints are satisfied whenever $x_{ij} = 0$. For $x_{ij} = 1$, we have:

$$u_i - u_j + n x_{ij} = (t) - (t+1) + n = n - 1,$$

satisfying the constraint.

Linear programming

- Similar to ILP (both constraints and objective are linear functions of the variables). However, for LP the variables are not restricted to be integers; they can be any real number. So not only are the domains infinite for each variable, they are *uncountably infinite*. Integer (and e.g., binary) variables are not allowed for LP.
  - Often there are nonnegativity constraints on some of the variables, e.g., Xi >= 0.
  - Cannot impose integrality constraints, e.g., for manufacturing problem could not use binary variables to ensure whole buildings are built, and may end up with solution such as yij=0.8, which is nonsensical (can't build 0.8 of a building).

LP vs ILP

- Which is easier to solve, LP or ILP?

LP vs. ILP

- Every LP is also an ILP (can just not include any integer variables), so clearly ILP is at least as hard as LP. It turns out that LP can be solved in polynomial-time, while ILP is NP-hard. In fact, several algorithms for ILP involve solving a series of LP "relaxations," where several of the integer variables are assigned to specific values and the resulting optimization formulation is solved as a linear program without any integrally-constrained variables.
- This is perhaps counterintuitive, as for LP variables all have infinite domain, but for ILP they may even just have domains of size 2.
- That said, of course huge LPs are more difficult to solve than tiny ILPs in practice, and worst-case complexity does not tell the full story.

ILP algorithms

- Exhaustive enumeration: can be performed if all variables have finite domain (can't be done if there are non-integral variables or integral variables over infinite domain). Can iterate over all possible combinations of variable values. For each combination, test for **feasibility** (whether it satisfies all constraints). If it is feasible, compute the objective value, and ultimately output the assignment that has highest objective value out of feasible solutions.
- Is this algorithm efficient?

ILP algorithm

- Unfortunately, the number of possible solutions is 2^n, where n is the number of variables. For n = 20, there are more than 1,000,000 candidates; for n=30, the number is greater than 1,000,000,000, which is too large to be solved by computers.

0-1 integer program example

$$\left.\begin{aligned} &\text{Maximize } z = \sum_{j=1}^{n} c_j x_j \\ &\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \le b_i, \quad i = 1, \ldots, m \\ &\qquad\qquad x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n \end{aligned}\right\}$$

ILP search tree

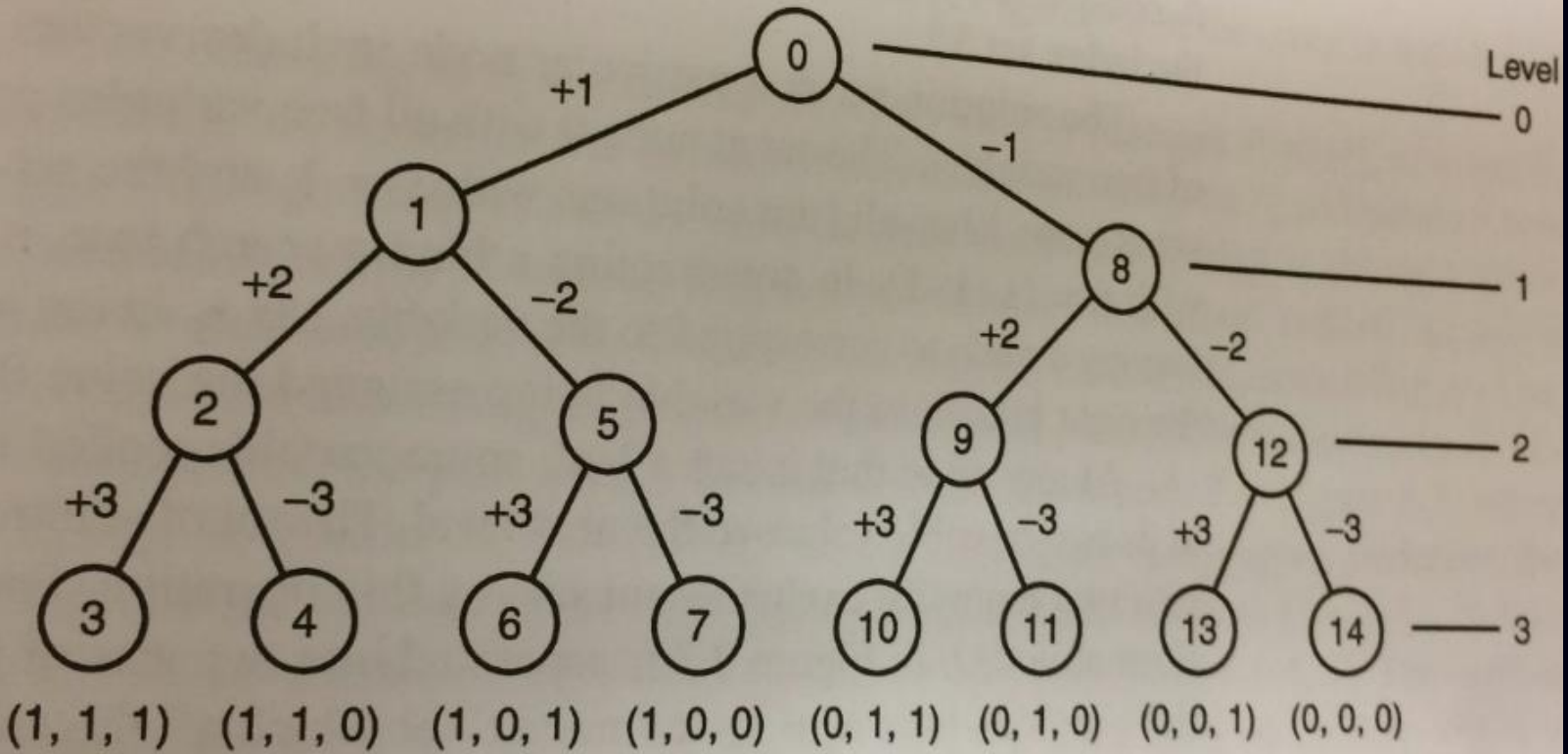


Figure 8. 3 Exhaustive search tree

ILP search tree

- We draw the tree with the *root* at the top and the *leaves* at the bottom. The circles are called *nodes*, and the lines are called *branches*. At the very top of the tree, we have node 0 or the root. As we descend the tree, decisions are made as indicated by the numbers on the branches. A negative number, -j, implies that the variable x_j has been set equal to 0, whereas a positive number, +j, implies that x_j has been set equal to 1.

ILP algorithm

- The nodes are numbered sequentially as the variables are fixed to either 0 or 1. The sequence will vary depending on the enumeration scheme. Each node k inherits all the restrictions defined by the branches on the path joining it to the root. This path is given the designation Pk. For example, at node 1 the decision +1 is indicated y the branch joining node 0 to node 1. This means we have set variable x1 equal to 1. At node 5, the decision -2 is indicated by the branch joining nodes 1 and 5, so we have the additional restriction x2 = 0. The leaves at the bottom of the tree signal that all variables have been fixed. Each of these eight nodes represents a complete solution that can be identified by tracing the path from the leaf node to the root and noting the decisions associated with the branches traversed along the way. Thus, node 6 represents the solution x = (1,0,1), whereas node 10 represents x = (0,1,1).

ILP algorithm

- Can perform a recursive DFS backtracking search algorithm (similar to both CSP backtracking search and minimax search) on this search tree.
- Could always branch to the left, arbitrary branching, or use more intelligent heuristics.
- Can integrate various pruning techniques like we did for minimax search (e.g., alpha-beta pruning) and for CSP search.

Branch and bound

- LP *relaxation*: the ILP but without the integrality constraints
- Suppose we have an *incumbent* solution with objective value zB, and zK is the objective value of the LP relaxation at node k.
- Four alternatives:
  - LP has no feasible solution (in which IP also has no feasible solution)
  - LP has an optimal solution with lower objective value (in which the current IP optimal solution is better than the LP optimal one and cannot provide an improvement over the incumbent).
  - Optimal solution to the LP is integer valued and feasible, and yields improved solution.
  - None of the above: i.e., the optimal LP solution improves the objective but is not integer-valued.
- For first 3 cases nothing more to be done. Only for case 4 is further branching needed.

Branch and bound

- Note that the *relaxed problem* associated with each node does not have to be an LP. A second choice could be an IP that is easier to solve than the original. Typical relaxations of the traveling salesman problem, for instance, are the assignment problem and the minimum spanning tree (MST) problem.

Branch and bound (B&B)

- We now elaborate and present the basic steps that are needed for solving a 0-1 integer program using B&B (can also be used for IPs with larger domains). Although most steps are general in that they are appropriate for a variety of problem classes, several computational procedures are problem dependent. Although a maximization objective is assumed, if the goal is to minimize, the problem can be solved with the same algorithm after making a few modifications, or directly by converting it to a maximization problem. The five routines below are used to guide the search for the optimal solution and to extract information that can be used to reduce the size of the B&B tree.

Branch and Bound

- *Bound*: This procedure examines the relaxed problem at a particular node and tries to establish a bound on the optimal solutions. It has two possible outcomes:
  1. An indication that there is no feasible solution in the set of integer solutions represented by the node
  2. A value z_{UB}– an upper bound on the objective for all solutions at the node and its descendent nodes

Branch and Bound

- *Approximate*: This procedure attempts to find a *feasible* integer solution from the solution of the relaxed problem. If one is found, it will have an objective value, call it Z_LB, that is a lower bound on the optimal solution for a maximization problem.
- *Variable fixing*: This procedure performs logical tests on the solution found at a node. The goal is to determine if any of the free binary variables are necessarily 0 or 1 in an optimal integer solution at the current node or at its descendants, or whether they must be set to 0 or 1 to ensure feasibility as the computations progress.

Branch and Bound

- *Branch*: A procedure aimed at selecting one of the free variables for separation. Also decided is the first direction (0 or 1) to explore.
- *Backtrack*: This is primarily a bookkeeping procedure that determines which node to explore next when the current node is fathomed. It is designed to enumerate systematically all remaining live nodes of the B&B tree while ensuring that the optimal solution to toe original IP is not overlooked.

Branch and bound algorithm

The following is the skeleton of a generic branch and bound algorithm for minimizing an arbitrary objective function f.[2] To obtain an actual algorithm from this, one requires a bounding function g, that computes lower bounds of f on nodes of the search tree, as well as a problem-specific branching rule.

1. Using a heuristic, find a solution x_h to the optimization problem. Store its value, $B = f(x_h)$. (If no heuristic is available, set B to infinity.) B will denote the best solution found so far, and will be used as an upper bound on candidate solutions.
2. Initialize a queue to hold a partial solution with none of the variables of the problem assigned.
3. Loop until the queue is empty:
    1. Take a node N off the queue.
    2. If N represents a single candidate solution x and $f(x) < B$, then x is the best solution so far. Record it and set $B \leftarrow f(x)$.
    3. Else, *branch* on N to produce new nodes N_i. For each of these:
        1. If $g(N_i) > B$, do nothing; since the lower bound on this node is greater than the upper bound of the problem, it will never lead to the optimal solution, and can be discarded.
        2. Else, store N_i on the queue.

Several different queue data structures can be used. A stack (LIFO queue) will yield a depth-first algorithm. A best-first branch and bound algorithm can be obtained by using a priority queue that sorts nodes on their g-value.[2] The depth-first variant is recommended when no good heuristic is available for producing an initial solution, because it quickly produces full solutions, and therefore upper bounds.[6]

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.

Nonlinear optimization

- Maximize (or minimize) f(x)

  subject to g_i(x) <= 0 for each i in {1,…,m}

  h_j = 0 for each j in {1,…,p)}

  x in X
- n,m,p positive integers
- X is subset of R^n (e.g., [0,1], or [-infinity,infinity]
- F, g_i, h_j real-valued functions on X for each I and each j, with at least one of f, g_i, h_j being *nonlinear*

Nonlinear optimization

- If the objective function f is linear and the constrained space is a polytope, the problem is a linear programming problem, which may be solved using well-known linear programming techniques such as the simplex method.
- If the objective function is concave (maximization problem), or convex (minimization problem) and the constraint set is convex, then the program is called convex and general methods from convex optimization can be used in most cases.
- If the objective function is quadratic and the constraints are linear, quadratic programming techniques are used.
- If the objective function is a ratio of a concave and a convex function (in the maximization case) and the constraints are convex, then the problem can be transformed to a convex optimization problem using fractional programming techniques.

Nonlinear optimization

- Several methods are available for solving nonconvex problems. One approach is to use special formulations of linear programming problems. Another method involves the use of branch and bound techniques, where the program is divided into subclasses to be solved with convex (minimization problem) or linear approximations that form a lower bound on the overall cost within the subdivision. With subsequent divisions, at some point an actual solution will be obtained whose cost is equal to the best lower bound obtained for any of the approximate solutions. This solution is optimal, although possibly not unique. The algorithm may also be stopped early, with the assurance that the best possible solution is within a tolerance from the best point found; such points are called ε-optimal. Terminating to ε-optimal points is typically necessary to ensure finite termination. This is especially useful for large, difficult problems and problems with uncertain costs or values where the uncertainty can be estimated with an appropriate reliability estimation.

Nonlinear programming

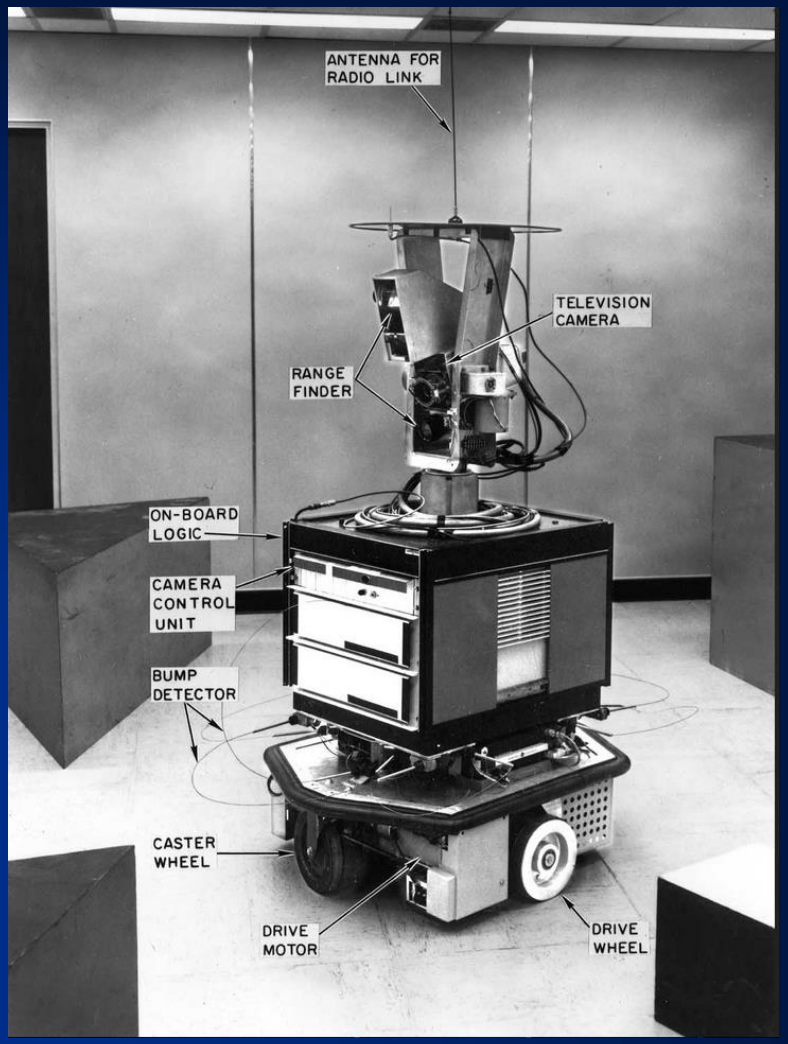
- Quadratic programming: For positive definite Q, the ellipsoid method solves the problem in polynomial time. If, on the other hand, Q is indefinite, then the problem is NP-hard. In fact, even if Q has only one negative eigenvalue, the problem is NP-hard.
- Convex optimization: variability complexity, often solved by *gradient* or *subgradient* methods.
- The following problems are all convex minimization problems, or can be transformed into convex minimizations problems via a change of variables: Least squares, Linear programming, Convex quadratic minimization with linear constraints, quadratic minimization with convex quadratic constraints, Conic optimization, Geometric programming, Second order cone programming, Semidefinite programming, Entropy maximization with appropriate constraints

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



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."

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

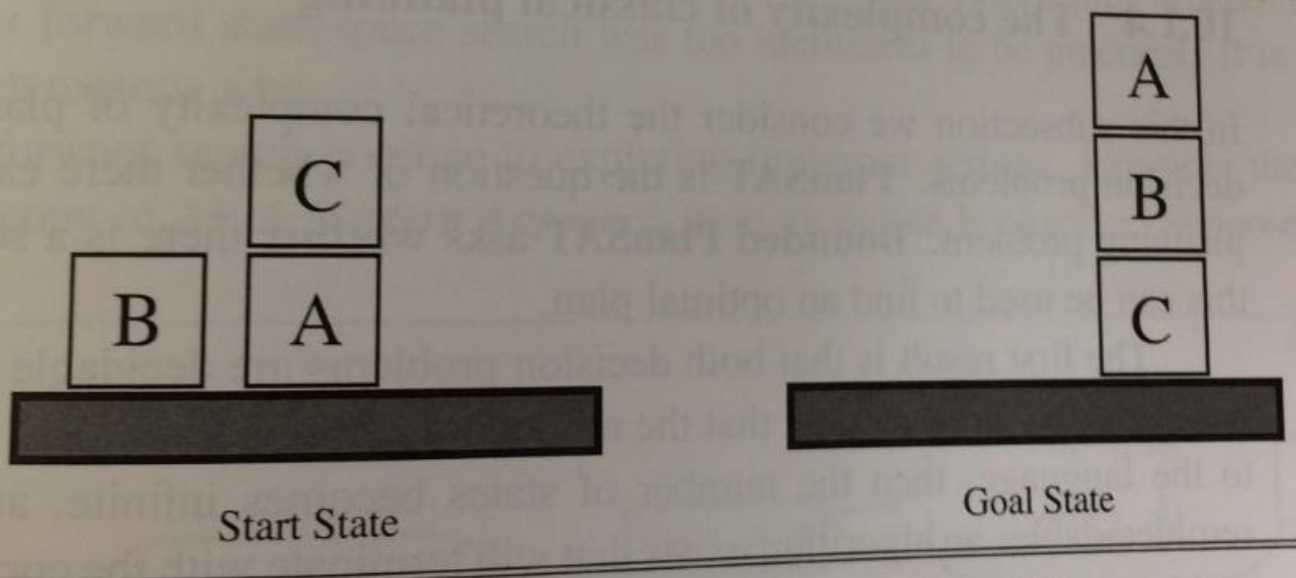


Figure 10.4 Diagram of the blocks-world problem in Figure 10.3.

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

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

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))

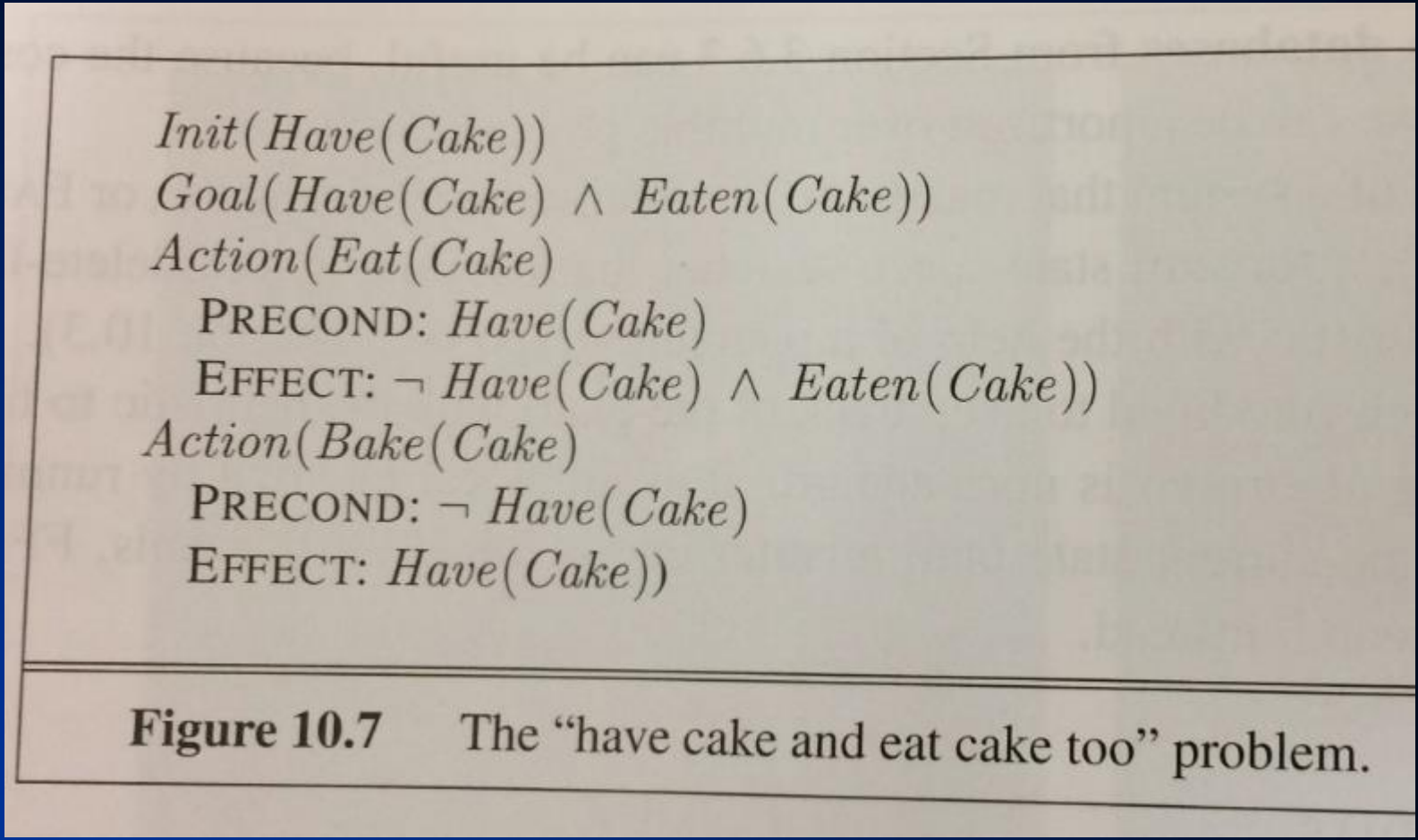
Blocks world

- 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

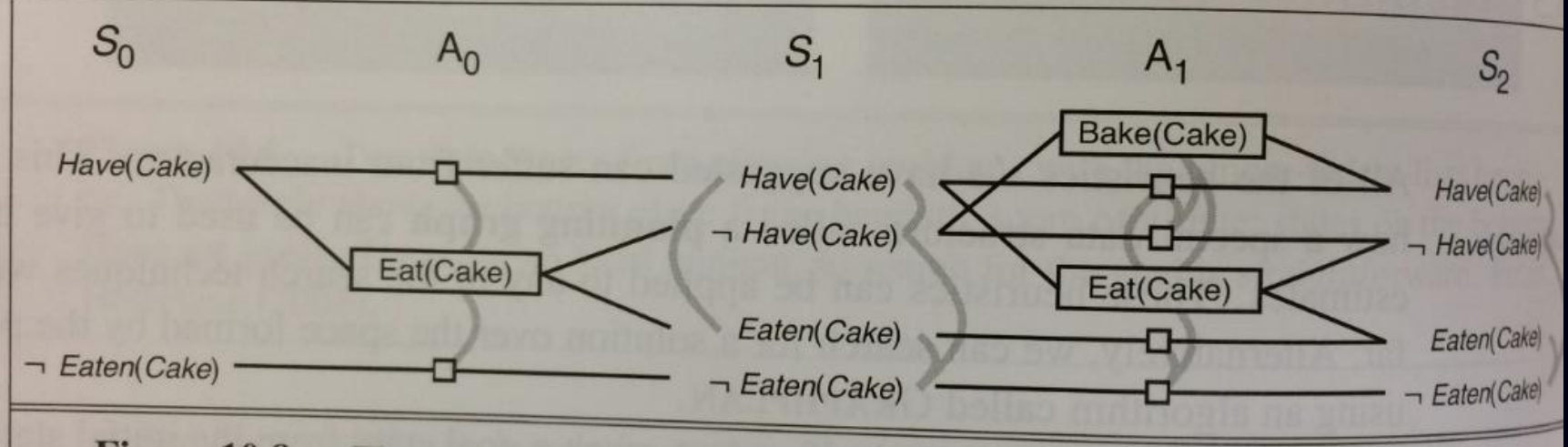


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because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

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.

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].

Homework for next class

- Chapter 13 from Russel/Norvig