CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

- http://www.ultimateaiclass.com/
- https://moodle.cis.fiu.edu/
- HW1 was due on Tuesday 10/3
  - Remember that you have up to 4 late days to use throughout the semester.
- HW2 out last week, due 10/17
- Midterm on 10/19
  - Covering search (uninformed, informed, local, adversarial, CSP), logic, and optimization
  - Review during half of class on 10/17

Upcoming lectures

- 10/5: Continue CSP
- 10/10: Wrap up CSP, start logic (propositional logic, first-order logic)
- 10/12: Wrap-up logic (logical inference), start optimization (integer, linear optimization)
- 10/17: Wrap up optimization (nonlinear optimization), midterm review
- 10/19: Midterm
- 10/24: TA will go over midterm and homework solutions in lecture
- Planning lecture will be after midterm on 10/26

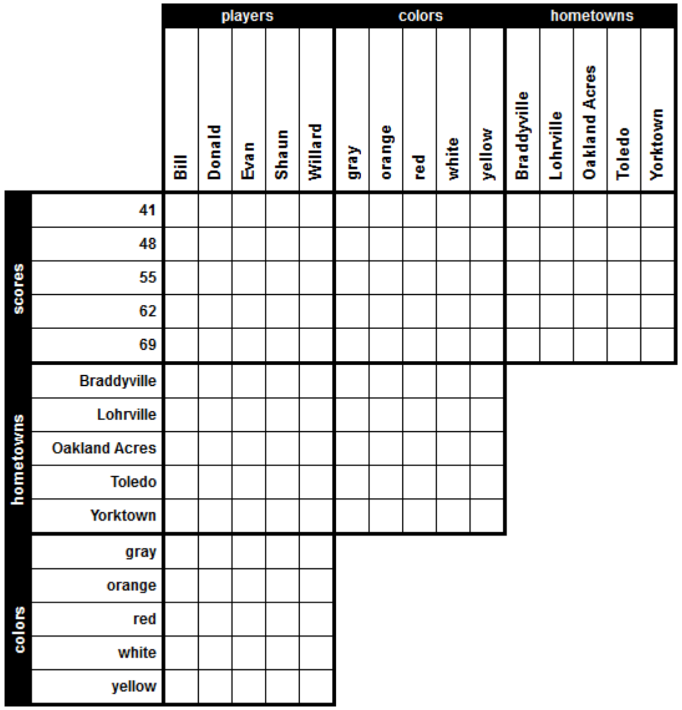
HW1

- Will be back before midterm
- Received 29 on moodle (33 students enrolled)
- Will be lenient regarding late days for HW1 due to the hurricane
  - 0-24 hours late: 0 late days
  - 1-3 days late: 1 late day
  - 3-5 days late: 2 late days
  - > 5 days late: 3 late days

HW2

- Due 10/17 at 2:05 in class (or 2pm on Moodle)
- Several exercises from textbook
- Logic puzzles that you must formulate models for as search/optimization problems using two different approaches (e.g., could be CSP, logical inference, integer programming). You can solve them using built-in Python solver libraries (e.g., for CSP and ILP) or build your own solver (possibly for extra credit). Open-ended question and many possible correct answers and approaches.
- http://www.logic-puzzles.org/

Quizzle



Hint | Clear Errors | Start Over

Clues | Notes | Answers

Active Clues

1. The player who threw the gray darts was either the player from Oakland Acres or Willard.

2. Evan threw the white darts.

3. Neither the player from Yorktown nor the contestant from Lohrville was Donald.

4. The contestant who scored 48 points wasn't from Oakland Acres.

5. Donald wasn't from Oakland Acres.

6. The contestant from Yorktown scored 7 points higher than the contestant who threw the gray darts.

7. The player who threw the red darts scored 7 points higher than Evan.

8. The contestant who threw the white darts scored 7 points higher than the player who threw the yellow darts.

9. The player who threw the orange darts finished somewhat lower than Willard.

10. Of Bill and the contestant who scored 41 points, one was from Braddyville and the other threw the orange darts.

Backstory And Goal

Lou's Bar and Grill held a friendly darts tournament this week. Using only the clues that follow, match each player to his score, hometown and dart color.

Remember, as with all grid-based logic puzzles, no option in any category will ever be used more than once. If you get stuck or run into problems, try the "Clear Errors" button to remove any mistakes that might be present on the grid, or the "Hint" button to see the next logical step in the puzzle.

CSP summary

- **Constraint satisfaction problems** represent a state with a set of variable-value pairs and represent the conditions for a solution by a set of constraints on the variables. Many real-world problems can be described as CSPs.
- A number of inference techniques use the constraints to infer which variable/value pairs are consistent and which are not. These include node, arc, path, and k-consistency.
- **Backtracking search**, a form of depth-first search, is commonly used for solving CSPs. Inference can be interwoven with search.
- The **minimum-remaining values** and **degree** heuristics are domain-independent methods for deciding which variable to choose next in a backtracking search. The **least-constraining value** heuristic helps in deciding which value to try first for a given variable. Backtracking occurs when no legal assignment can be found for a variable. **Conflict-directed backjumping** backtracks directly to the source of the problem.
- Local search using the **min-conflicts** heuristic has also been applied to constraint satisfaction problems with great success.

Logical agents

- The problem-solving (search) agents "know things," but only in a very limited, inflexible sense. For example, the transition model for the 8-puzzle—knowledge of what the actions do—is hidden inside the domain-specific code of the RESULT function. It can be used to predict the outcome of actions but not to deduce that two tiles cannot occupy the same space or that states with odd parity cannot be reached from states with even parity, etc. The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, an agent's only choice for representing what it knows about the current state is to list all possible concrete states—a hopeless prospect in large environments.

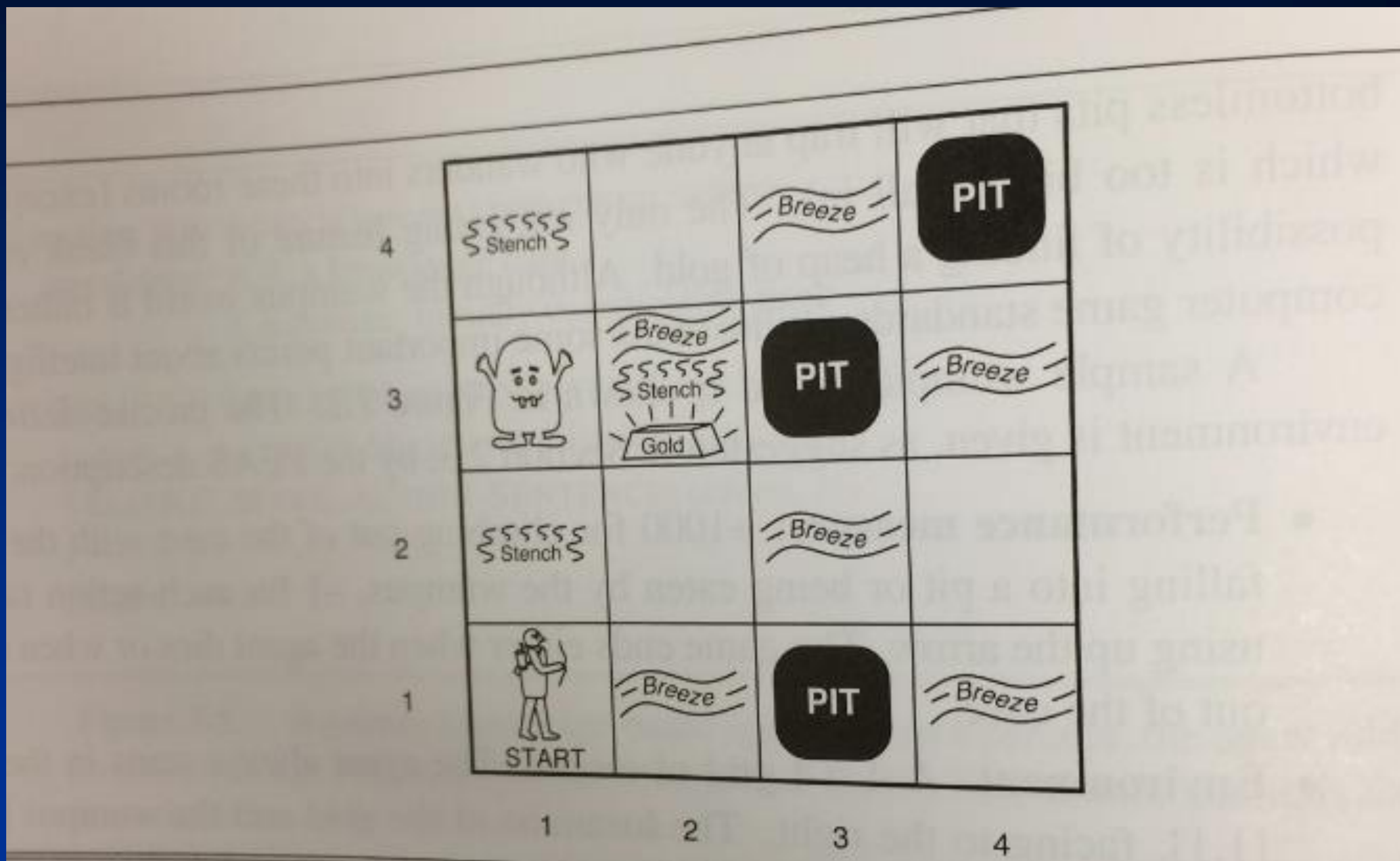
Logical agents

- Constraint satisfaction introduced the idea of representing states as assignments of values to variables; this is a step in the right direction, enabling some parts of the agent to work in a domain-independent way and allowing for more efficient algorithms. We now take this step to its logical conclusion—we develop **logic** as a general class of representations to support knowledge-based agents. Such agents can combine and recombine information to suit myriad purposes. Often this process can be quite far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the earth's life expectancy. Knowledge-based agents can accept new tasks in the form of explicitly-described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

Logical agents

- The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only mitigating feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

Wumpus world



Wumpus world

- **Performance measure**: +1000 for climbing out of the cave with the gold, -1000 for falling into a pit or being eaten by the wumpus, -1 for each action taken and -10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
- **Environment**: A 4x4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.

Wumpus world

- **Actuators**: The agent can move *Forward*, *TurnLeft* by 90 degrees, or *TurnRight* by 90 degrees. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].

Wumpus world

- **Sensors**: The agent has five sensors, each of which gives a single bit of information:
  - In the square containing the wumpus and in the directly (not diagonally) adjacent squares, the agent will perceive a *Stench*.
  - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
  - In the square where the goal is, the agent will perceive a *Glitter*.
  - When an agent walks into a wall, it will perceive a *Bump*.
  - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.
- The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench*, *Breeze*, *None*, *None*, *None*].

Wumpus world

- Consider a knowledge-based wumpus agent exploring the environment in the Figure 7.2. We use an informal knowledge representation language consisting of writing down symbols in a grid. The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively in square [1,1].
- The first percept is [*None*,*None*,*None*,*None*,*None*], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 7.3a shows the agent's state of knowledge at this point.

Wumpus world

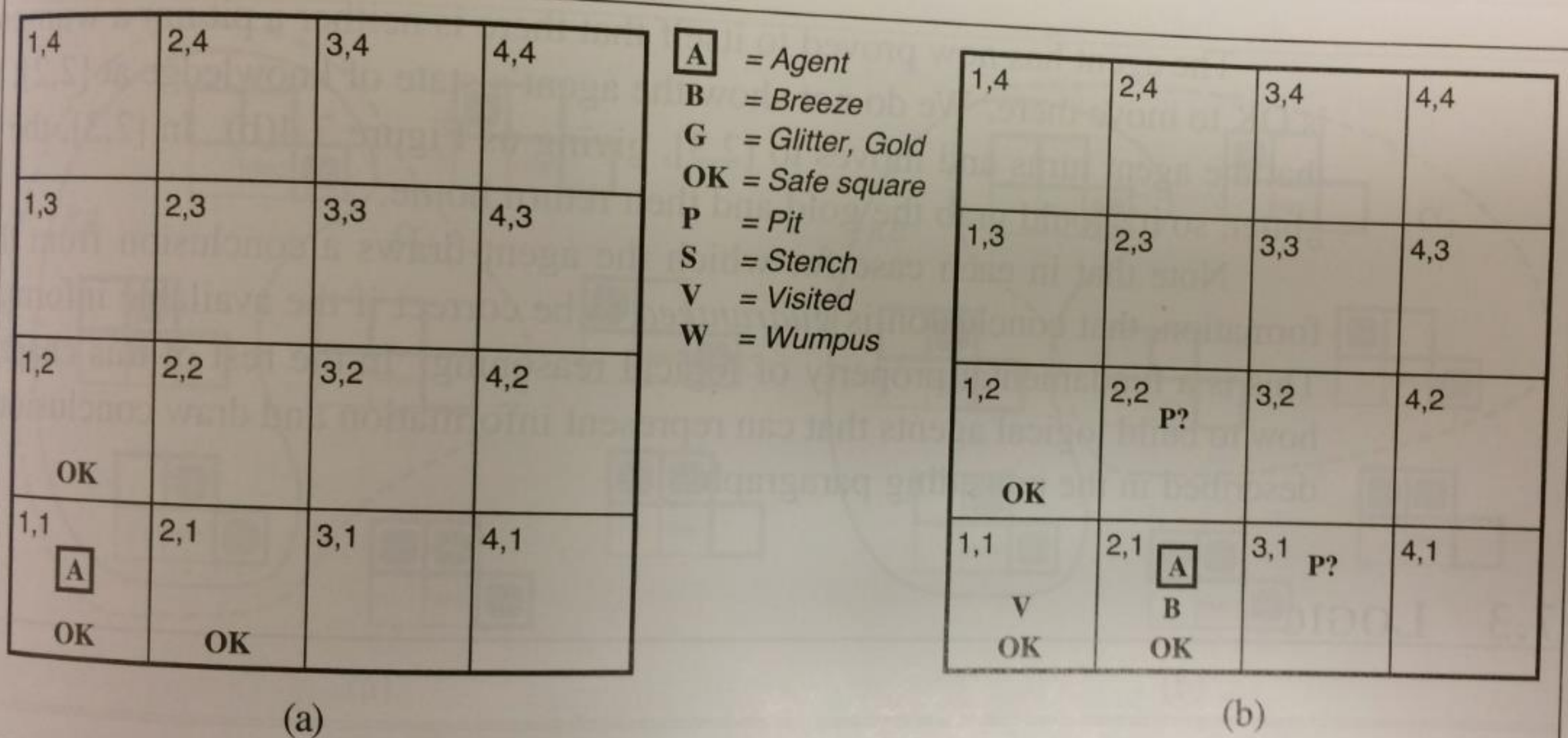


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

Wumpus world

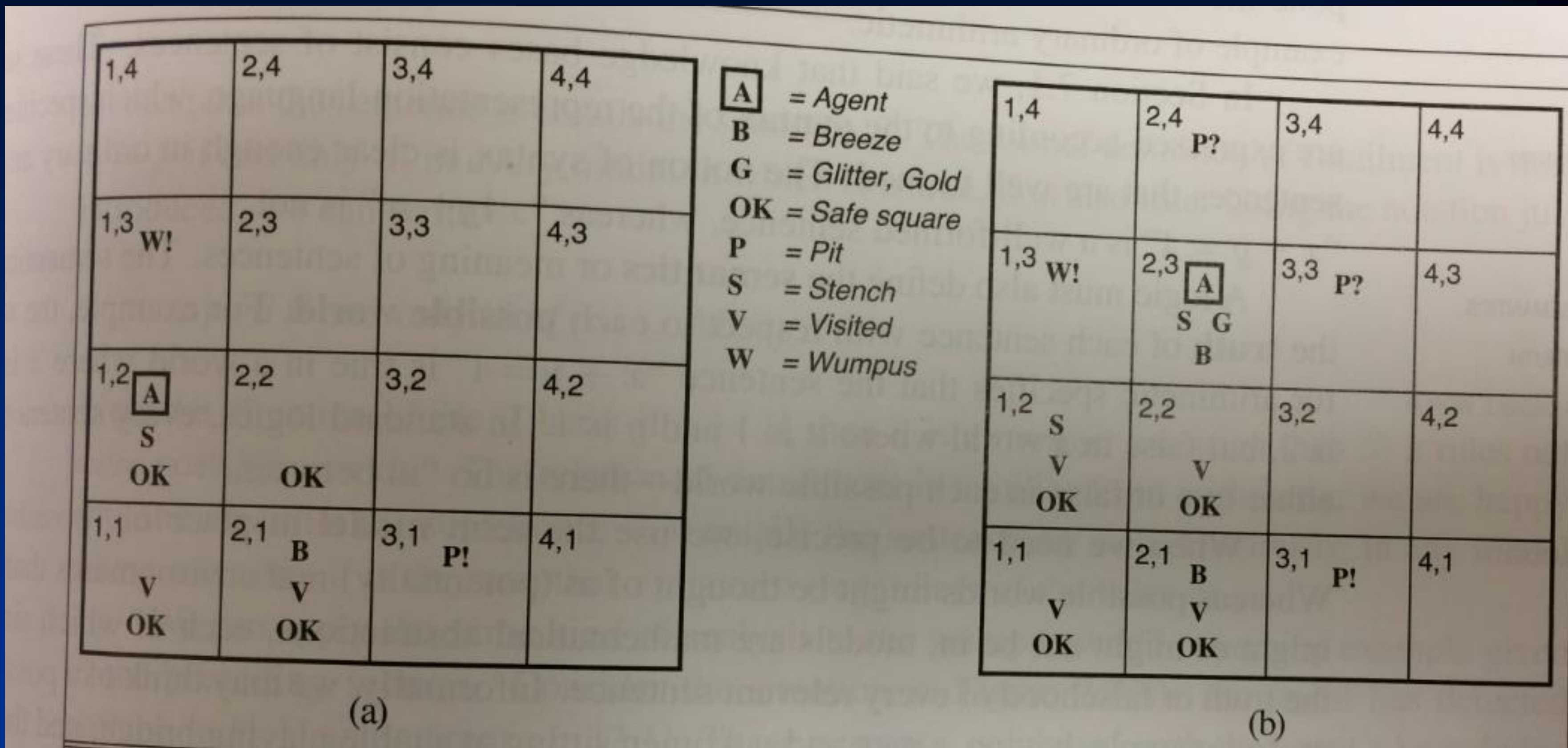


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

Wumpus world

- A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by “B”) in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation “P?” indicates a possible pit in those squares. At this point, there is only one known square that is OK and that as not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

Wumpus world

- The agent perceives a stench in [1,2], resulting in the state of knowledge shown in 7.4a. The stench in [1,2], means that there must be a wumpus nearby. But the wumpus cannot be in [1,1[, by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

Wumpus world

- The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us 74b. In [2,3], the agent detects a glitter, so it should grab the gold and then return home.
- Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning.

Logic

- Consider the situation in 7.3b: the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the **knowledge base** (KB). The agent is interested (among other things) in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might not contain a pit, so (for the purposes of this example) there are 2^3=8 possible **models**. These eight models are shown in 7.5.

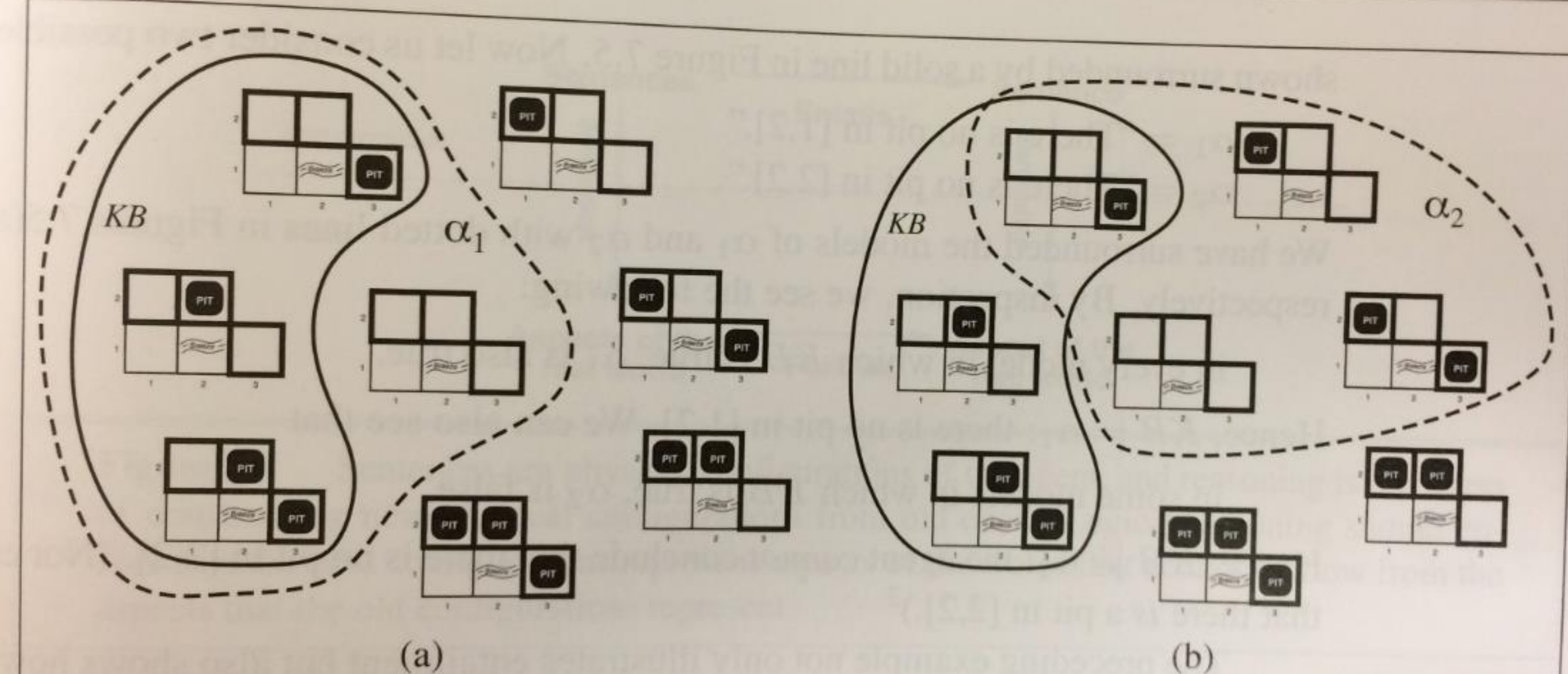


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

Logical agents

- The KB can be thought of a set of **sentences** or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows—for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are shown surrounded by a solid line in 7.5. Now let us consider two possible conclusions:
  - A1 = "There is no pit in [1,2]"
  - A2 = "There is no pit in [2,2]"
- A1 and A2 are surrounded with dotted lines in 7.5a and 7.5b. By inspection, we see the following:
  - In every model in which KB is true, A1 is also true.

Logical agents

- Hence, KB |= A1; there is no pit in [1,2]. We can also see that
  - In *some* models in which KB is true, A2 is false.
- Hence, KB !|= A2; the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)

Logical agents

- The preceding example not only illustrates **entailment** (i.e., one sentence following logically from another) but also shows how the definition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**. The inference algorithm in Figure 7.5 is called **model checking**, because it enumerates all possible **models** (i.e., possible "worlds") to check that alpha is true in all models in which KB is true, that is, that M(KB) is a subset of M(alpha).

Propositional logic

$$
\begin{array}{rcl}
\mathit{Sentence} & \rightarrow & \mathit{AtomicSentence} \mid \mathit{ComplexSentence} \\
\mathit{AtomicSentence} & \rightarrow & \mathit{True} \mid \mathit{False} \mid P \mid Q \mid R \mid \ldots \\
\mathit{ComplexSentence} & \rightarrow & (\ \mathit{Sentence}\ ) \mid [\ \mathit{Sentence}\ ] \\
 & \mid & \neg\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \land \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \lor \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Rightarrow\ \mathit{Sentence} \\
 & \mid & \mathit{Sentence} \ \Leftrightarrow\ \mathit{Sentence} \\
\textsc{Operator Precedence} & : & \neg, \land, \lor, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Propositional logic

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

Wumpus world

- Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We use the following symbols for each [x,y] location:
  - Pxy is true if there is a pit in [x,y]
  - Wxy is true if there is a wumpus in [x,y], dead or alive
  - Bxy is true if the agent perceives a breeze in [x,y]
  - Sxy is true if the agent perceives a stench in [x,y]

Wumpus world

- The sentences we write will suffice to derive !P12 (there is no pit in P12), as was done informally before. We label each sentence Ri so that we can refer to them:
  - There is no pit in [1,1]: R1 : !P11
  - A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:
    - R2: B11 <-> (P12 V P21)
    - R3: B21 <-> (P11 V P22 V P31)
  - The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3b:
    - R4: !B11, R5: B21

Wumpus world

- Our goal now is to decide whether KB |= A for some sentence A. For example, is !P12 entailed by our KB? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that A is true in every model in which HV is true. Models are assignments of *true* or *false* to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are B11,B21,P11,P12,P21,P22, and P31. With seven symbols, there are 2^7=128 possible models; in three of these, KB is true (Figure 7.9(. In those three models, !P12 is true, hence there is no pit in [1,2]. On the other hand, P2,2 is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2].

Wumpus world

- Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is in Figure 7.10. Like the BACKTRACKNIG-SEARCH algorithm for CSP, TT-ENTAILS? Performs a recursive enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements direction the definition of entailment, and **complete** because it works for any KB and A and always terminates—there are only finitely many models to examine.

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].

Logical inference algorithm

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })
```

```
function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
     if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
     else return true // when KB is false, always return true
  else do
     P ← FIRST(symbols)
     rest ← REST(symbols)
     return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
             and
             TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword "**and**" is used here as a logical operation on its two arguments, returning *true* or *false*.

Constraint satisfaction problems

- A constraint satisfaction problem consists of three components, X, D, and C:
  - X is a set of variables, $\{X_1,\ldots,X_n\}$.
  - D is a set of domains, $\{D_1,\ldots,D_n\}$, one for each variable.
  - C is a set of constraints that specify allowable combinations of values.

Example problem: Map coloring

- Suppose that, having tired of Romania, we are looking at a map of Australia showing each of its states and territories. We are given the task of coloring each region either red, green, or blue in such a way that no neighboring regions have the same color.
- To formulate this as a CSP, we define the variables to be the regions: X = {WA, NT, Q, NSW, V, SA, T}
- The domain of each variable is the set D_i = {red, green, blue}.
- The constraints require neighboring regions to have distinct colors. Since there are nine places where regions border, there are nine constraints: C = {SA!=WA, SA!=NT,SA!=Q, etc.}
- SA!=WA is shortcut for ((SA,WA),SA!=WA), where SA!=WA can be fully enumerated in turn as {(red,green),(red,blue),…}

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 $y_{ij} = 1$ if design i is used at site j, and 0 otherwise
- Also, denote by p_{ij} the annual net income and by a_{ij} 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

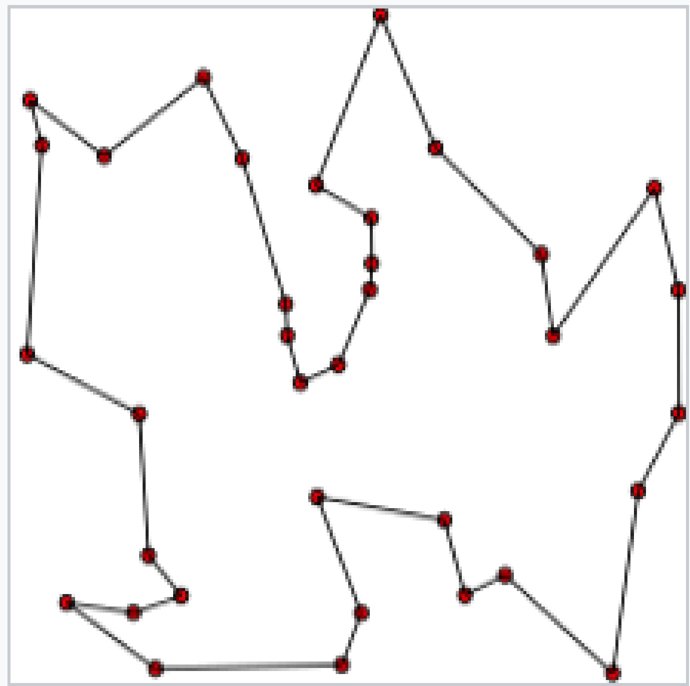
Manufacturing site selection

- 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:

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

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:

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

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

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

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?

Nonlinear programming

- Quadratic objective?
- Quadratic constraints?
- Cubic objective?
- Conic objective?
- Arbitrary objective and constraints (like CSP)?

Homework for next class

- Chapters 25 from Russel/Norvig
- HW2: due 10/17 at 2:05 in class (or 2pm on Moodle)