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 10/24
  - Covering search (uninformed, informed, local, adversarial, constraint satisfaction), logic, and optimization

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: Continue optimization (integer, linear optimization)
- 10/19: Wrap up optimization (nonlinear optimization), go over homework 1 (and parts of hw2 if all students have turned it in by start of class), midterm review
- 10/24: Midterm
- 10/26: Planning

HW2

- Due 10/17 at 2:05 in class (or 2pm on Moodle)
- It is ok to work with one partner for homework 2. Must include document stating whom you worked with and describe extent of collaboration. This policy may be modified for future assignments.
- Remember late day policy.

Minesweeper

- Minesweeper is NP-complete
- http://simon.bailey.at/random/kaye.minesweeper.pdf

NP-completeness

- Consider Sudoku, an example of a problem that is easy to verify, but whose answer may be difficult to compute. Given a partially filled-in Sudoku grid, of any size, is there at least one legal solution? A proposed solution is easily verified, and the time to check a solution grows slowly (polynomially) as the grid gets bigger. However, all known algorithms for finding solutions take, for difficult examples, time that grows exponentially as the grid gets bigger. So Sudoku is in **NP** (quickly checkable) but does not seem to be in **P** (quickly solvable). Thousands of other problems seem similar, fast to check but slow to solve. Researchers have shown that a fast solution to any one of these problems could be used to build a quick solution to all the others, a property called **NP**-completeness. Decades of searching have not yielded a fast solution to any of these problems, so most scientists suspect that none of these problems can be solved quickly. However, this has never been proved.

Computational complexity

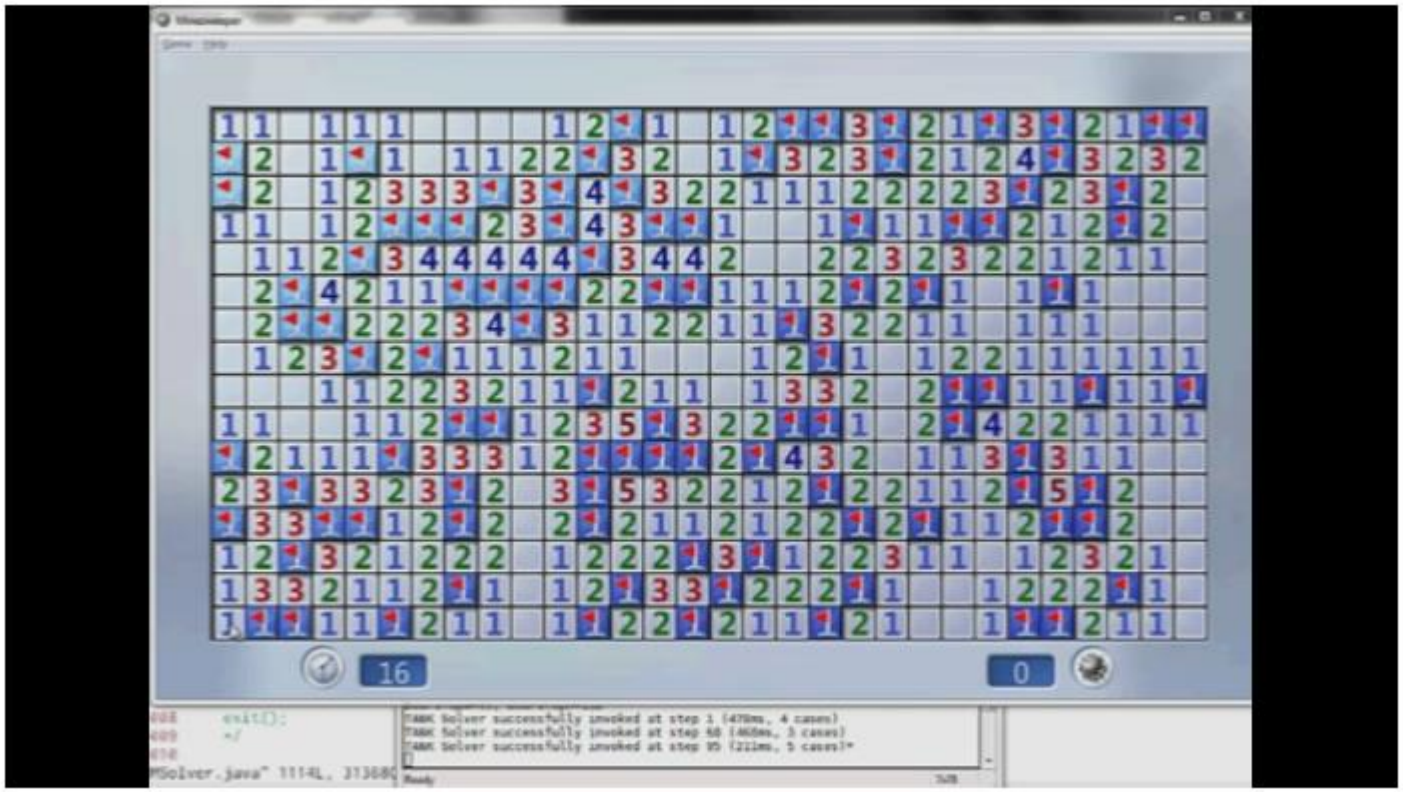
- P: polynomial-time algorithm exists
  - E.g., O(n), O(n^2), O(n^3), etc.
  - This is “efficient”
  - Most search algorithms we saw were NOT polynomial time
  - Many important AI problems can NOT be solved exactly in polynomial time
  - Theory does not always equal practice (e.g., poker, linear programming)
  - Polynomial-time algorithm can have constant in exponent, but no parameters in exponent.

NP

- NP: given a candidate solution, it can be verified in polynomial time whether it is actually a solution.
  - E.g., given a coloring of Australia map, can verify easily whether every pair of adjacent regions is a different color
- P is a subset of NP
- NP can also include many problems for which no polynomial-time algorithms are known
  - E.g., Sudoku, Minesweeper, integer programming
- Often the best-known algorithm runtime *exponential* in one or more parameters
  - E.g., for DFS it is O(b^m).
- P vs. NP problem: does there exist a polynomial-time algorithm for every problem in NP?

Minesweeper AI?

- https://luckytoilet.wordpress.com/2012/12/23/2125/



Straightforward algorithm

- “When the number 1 has exactly one empty square around it, then we know there’s a mine there.”
- “If a 1 has a mine around it, then we know that all the other squares around the 1 *cannot* be mines.”
- These two inference rules are good enough to solve beginner grid
- “The trivially straightforward algorithm is actually good enough to solve the beginner and intermediate versions of the game a good percent of the time. Occasionally, if we’re lucky, it even manages to solve an advanced grid!”

Tank Solver Algorithm

- “From the lower 2, we know that one of the two circled squares has a mine, while the other doesn’t. We just don’t know which one has the mine … Although this doesn’t tell us anything right now, we can combine this information with the next 2: we can deduce that the two yellowed squares are empty:”
- The idea for the Tank algorithm is to **enumerate all possible** configurations of mines for a position, and see what’s in common between these configurations.

Minesweeper AI

- Will Tank Solver Algorithm always work?

Minesweeper

- No, sometimes we will need to “guess.”
  - This is the same idea behind inference vs. search for CSP and logic.

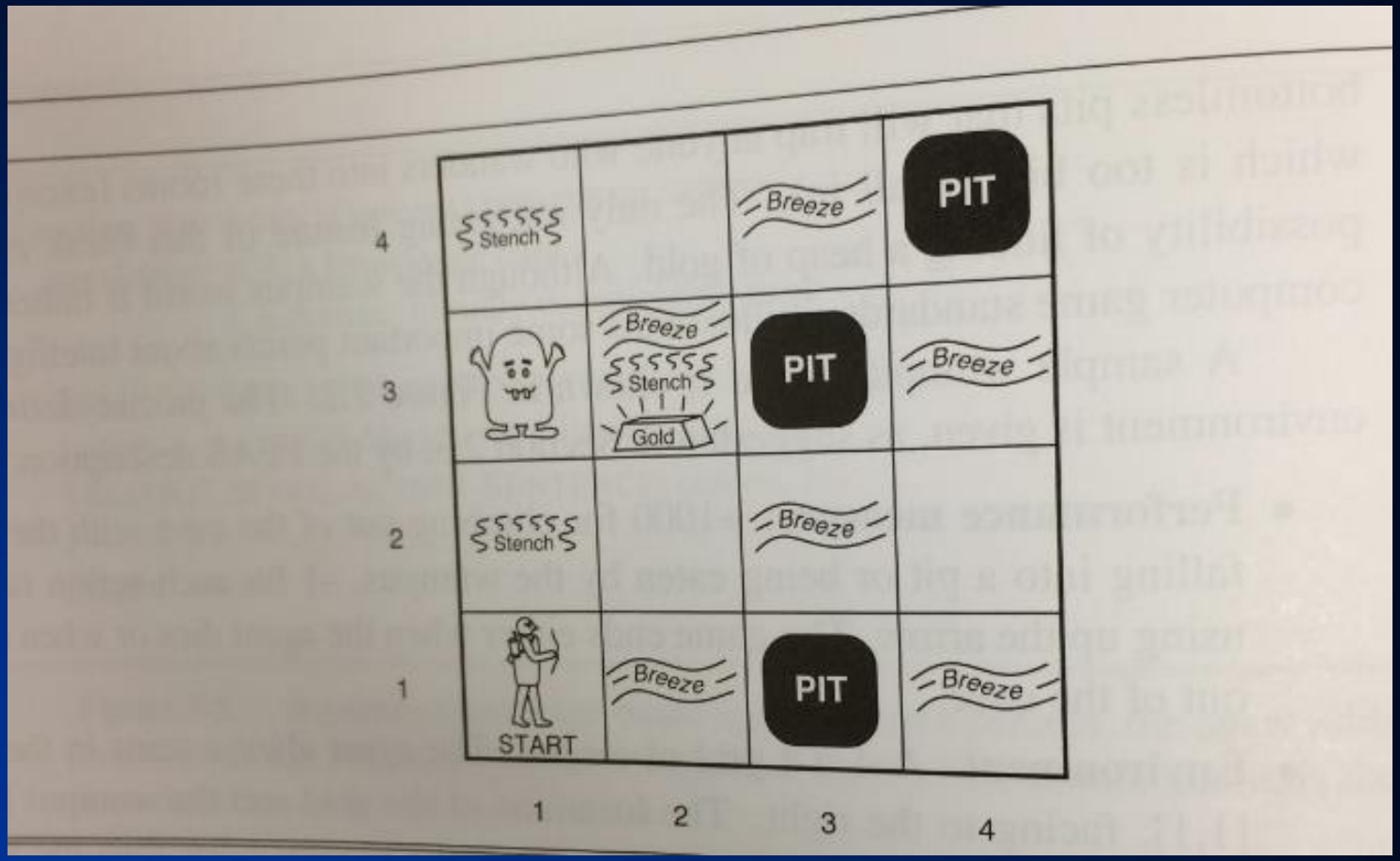
Minesweeper AI

- Two endgame tactics:
  - "what if the mine counter reads 1? The 2-mine configuration is eliminated, leaving just one possibility left. We can safely open the three tiles on the perimeter."
  - "The mine counter reads 2. Each of the two circled regions gives us a 50-50 chance – and the Tank algorithm stops here. Of course, the middle square is safe! To modify the algorithm to solve these cases, when there aren't that many tiles left, do the recursion on *all* the remaining tiles, not just the border tiles."

How do the algorithms do?

- Experiments on advanced grid:
- The naïve algorithm could not solve it, unless we get very lucky.
- Tank Solver with probabilistic guessing solves it about 20% of the time.
- Adding the two endgame tricks bumps it up to a 50% success rate.

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

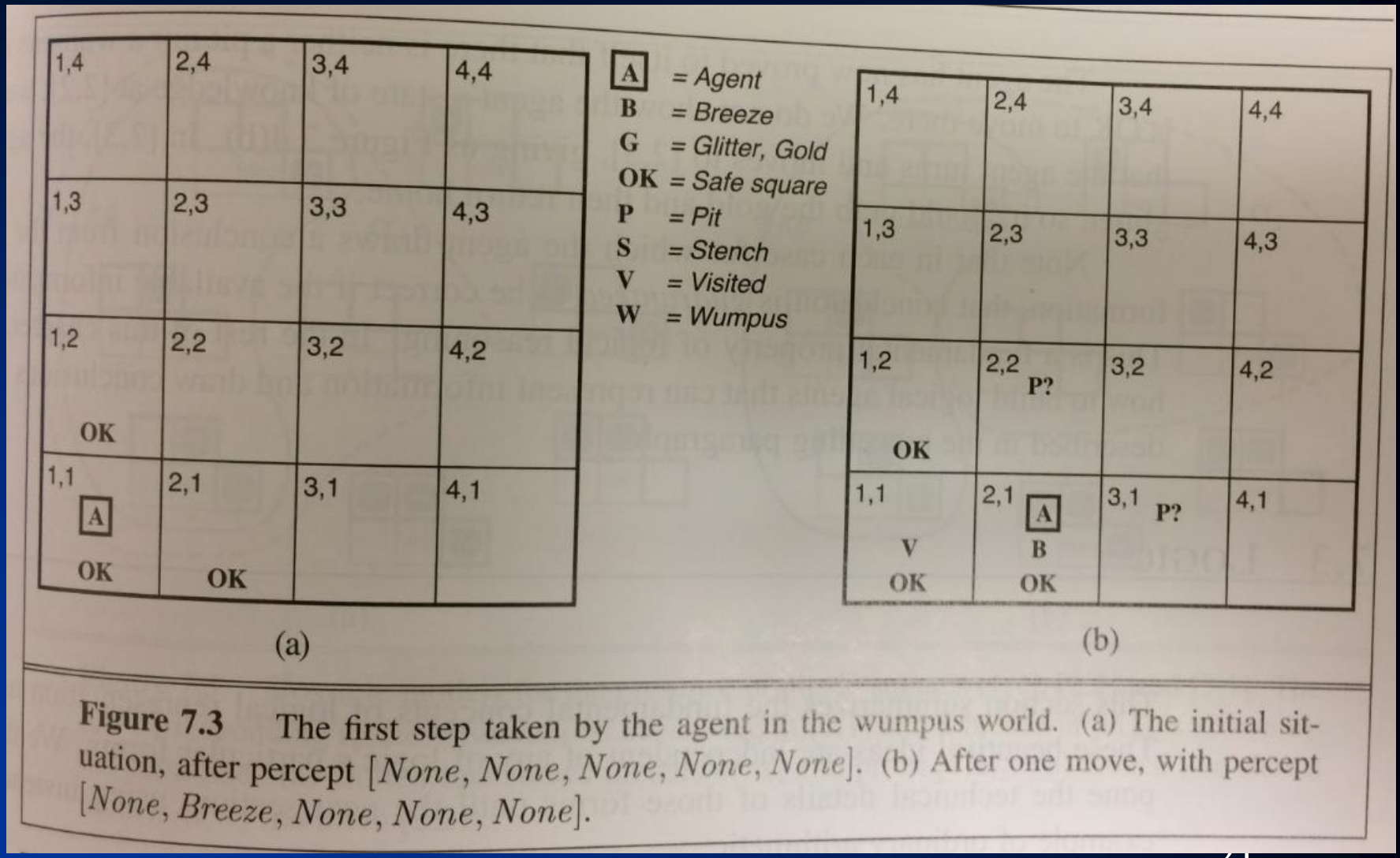


Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None*, *None*, *None*, *None*, *None*]. (b) After one move, with percept [*None*, *Breeze*, *None*, *None*, *None*].

Wumpus world

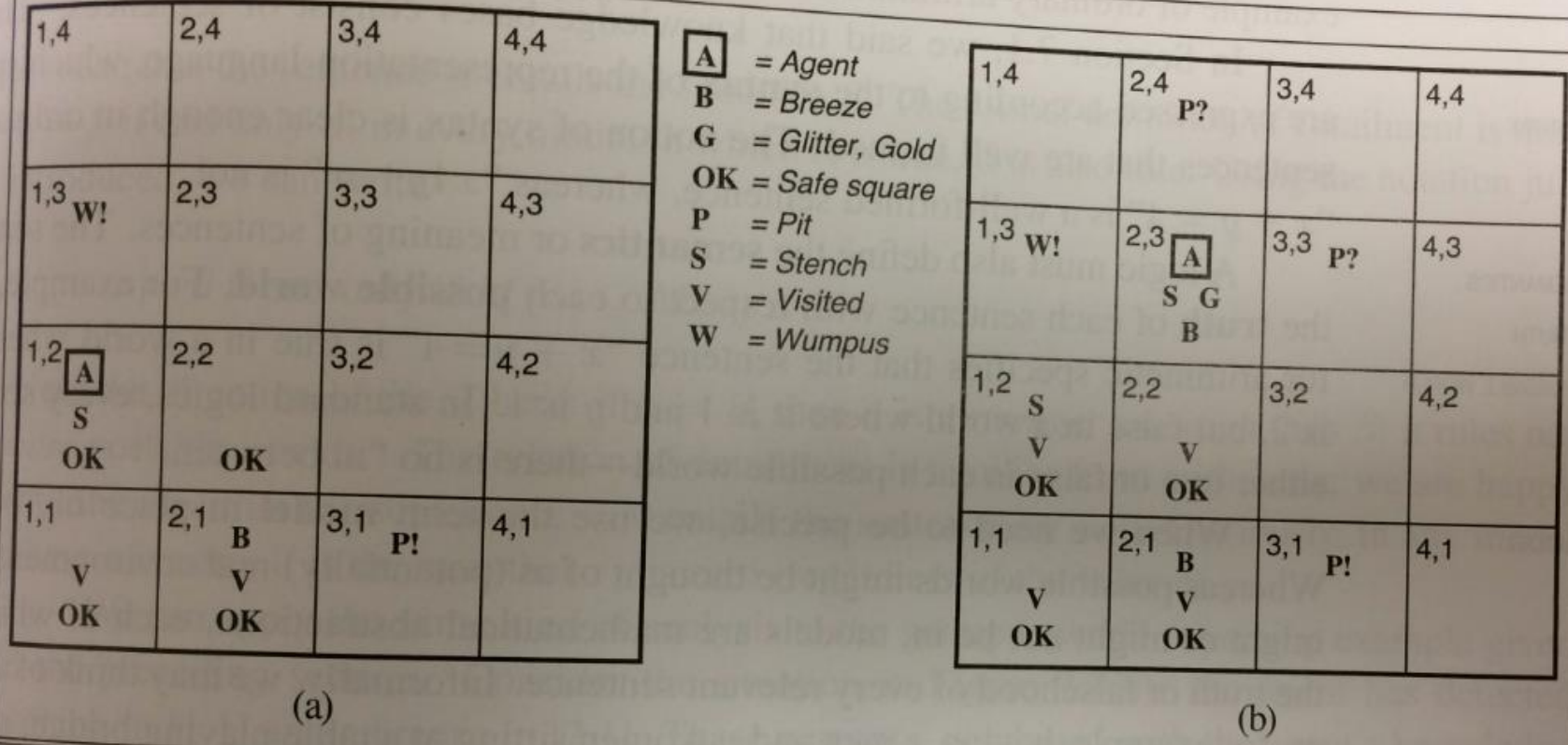


Figure 7.4 Two later stages in the progress of the agent. (a) After the third move, with percept [*Stench*, *None*, *None*, *None*, *None*]. (b) After the fifth move, with percept [*Stench*, *Breeze*, *Glitter*, *None*, *None*].

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.

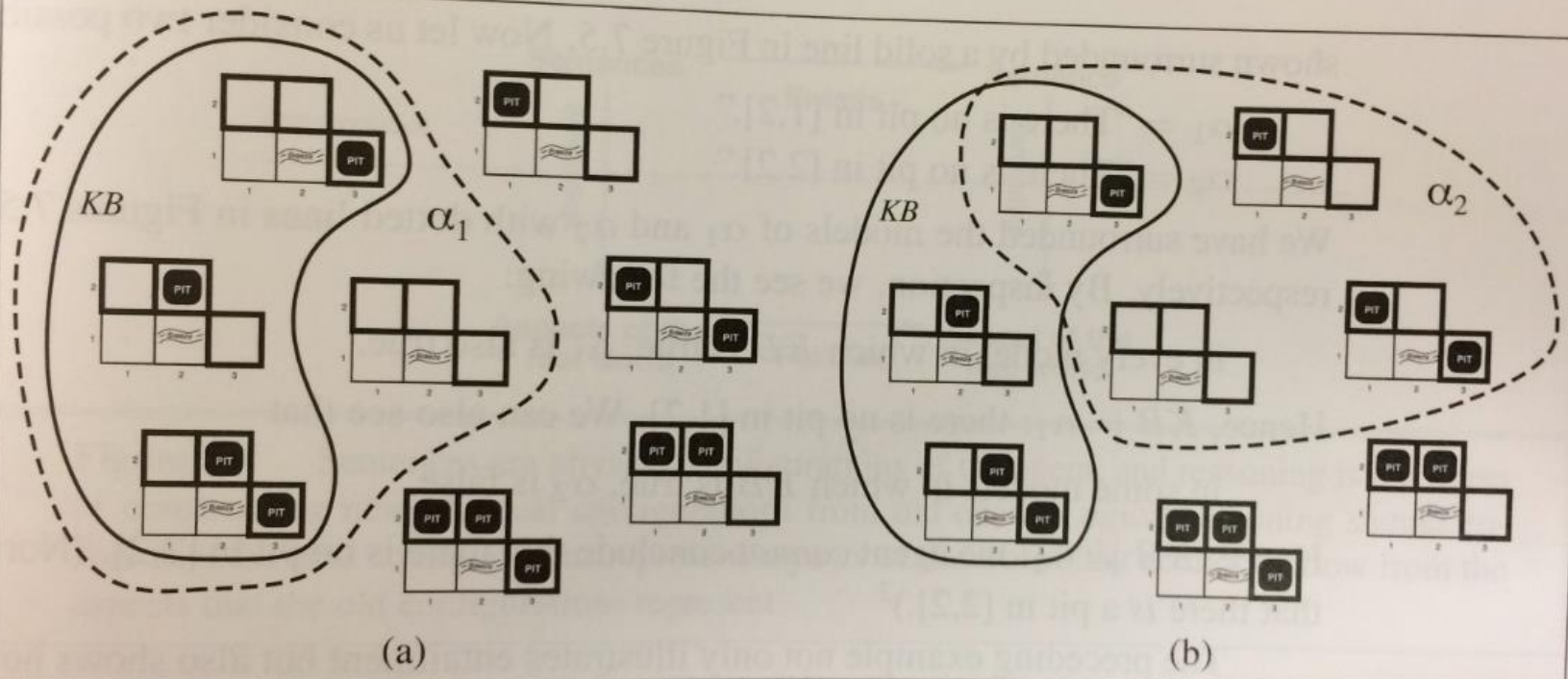


Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in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

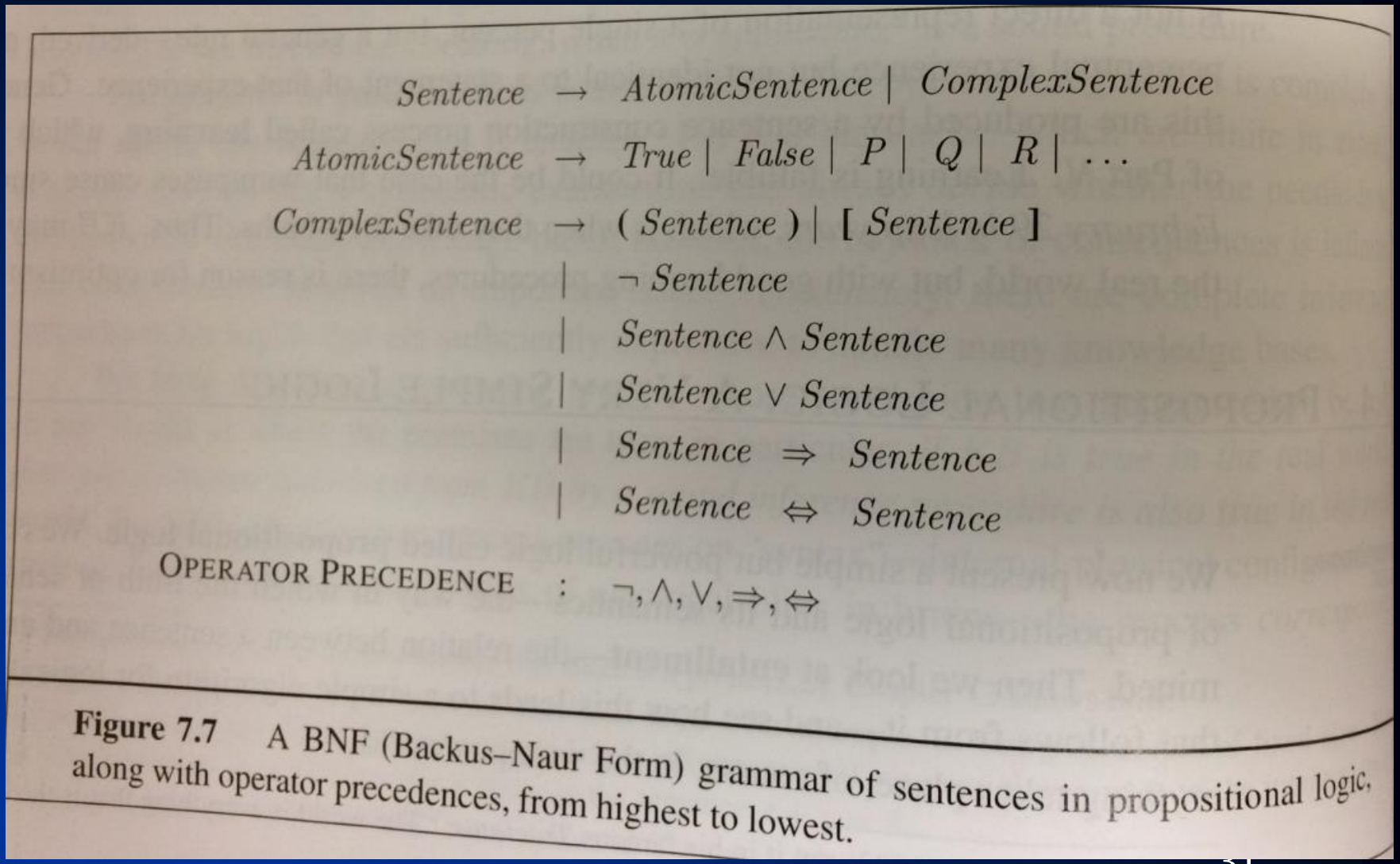
Logical agents

- 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



$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

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, P22 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 directly 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.

Remember 4 criteria for search algorithms

- Completeness
  - If a solution exists, the algorithm will find it
- Optimality
- Running time
- Space requirement

- Soundness is converse of completeness: a logical inference algorithm is **sound** if it derives only **entailed** sentences.
  - In general, a search algorithm is sound if the following holds: If no solution exists, the algorithm will output that there is no solution (it will not output a false "solution").

Soundness vs. completeness

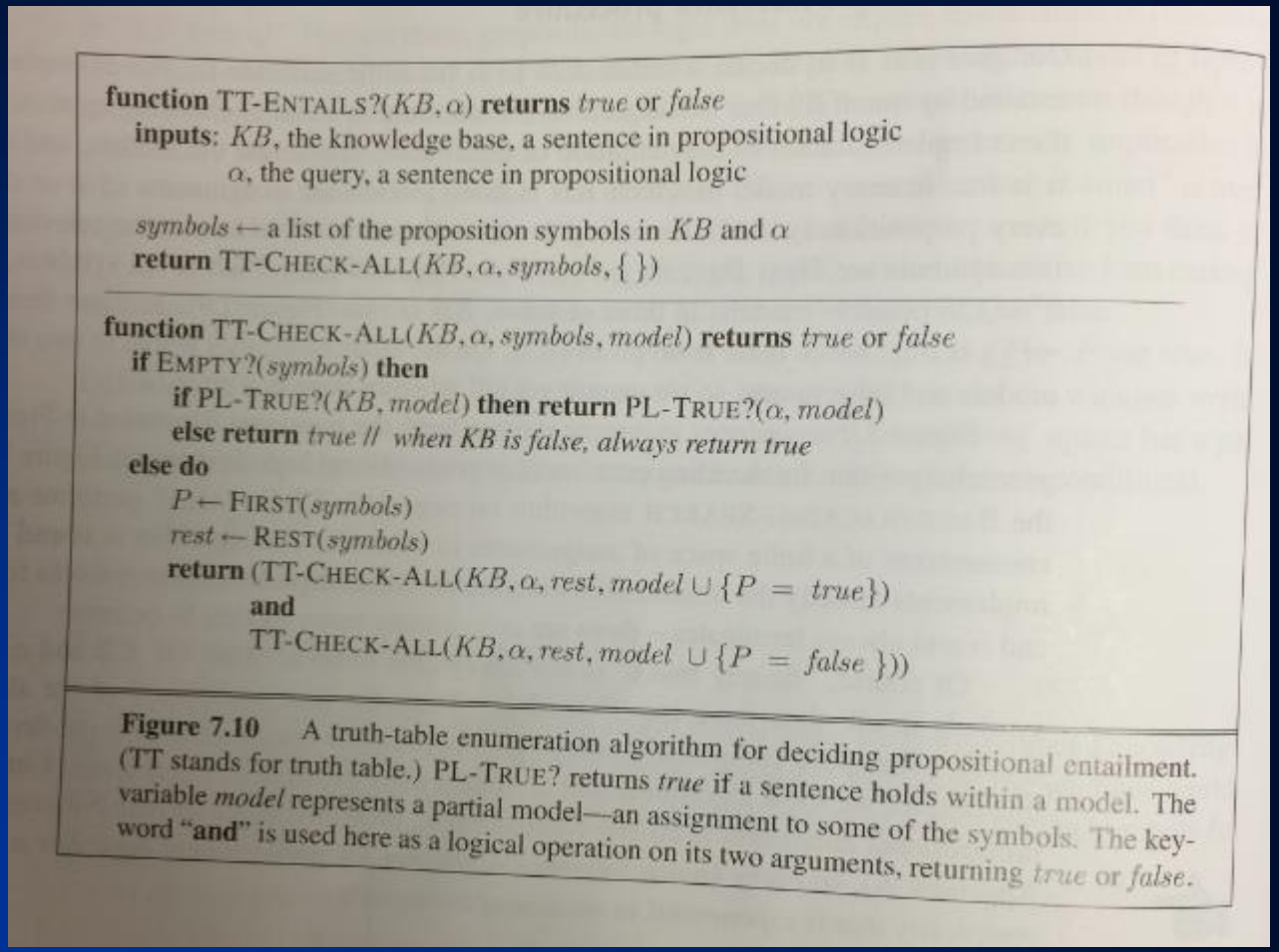
- Recall that a sentence A is **entailed** by sentence B if it follows logically from it using one of the rules we have seen.
- **Soundness** is highly desirable for logical inference: an unsound inference procedure "essentially makes things up as it goes along—it announces the discovery of nonexistent needles" and can derive some statements that are not logically implied by the knowledge base.
- **Completeness** property is also highly desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack.
- These are both important for general search as well.

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 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 = $\text{sum}_i \text{sum}_j$ pij yij
- Subject to:
  - $\text{sum}_i \text{sum}_j$ 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

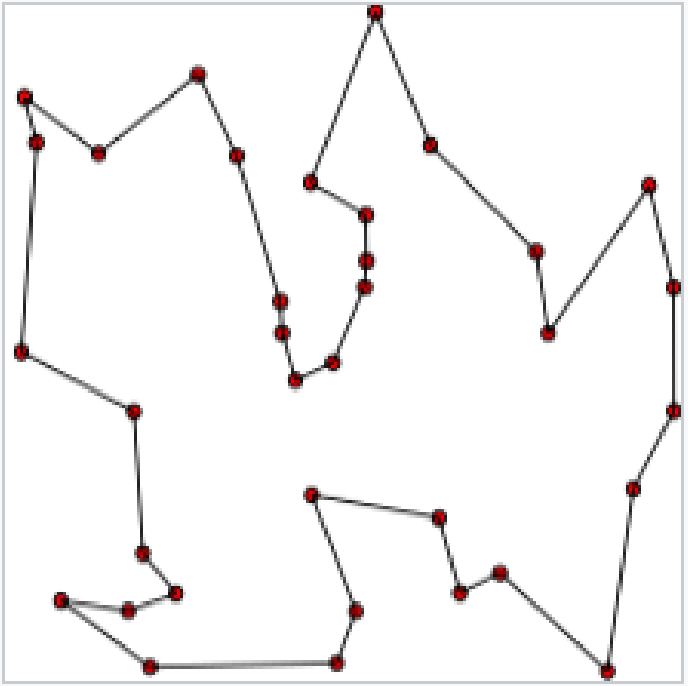
Manufacturing site selection

- 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:

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

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.

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

$$\text{Maximize } z = \sum_{j=1}^{n} c_j x_j$$

$$\left.\begin{array}{ll} \text{subject to} & \displaystyle\sum_{j=1}^{n} a_{ij} x_j \le b_i, \quad i = 1, \ldots, m \\ & x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n \end{array}\right\}$$

ILP search tree

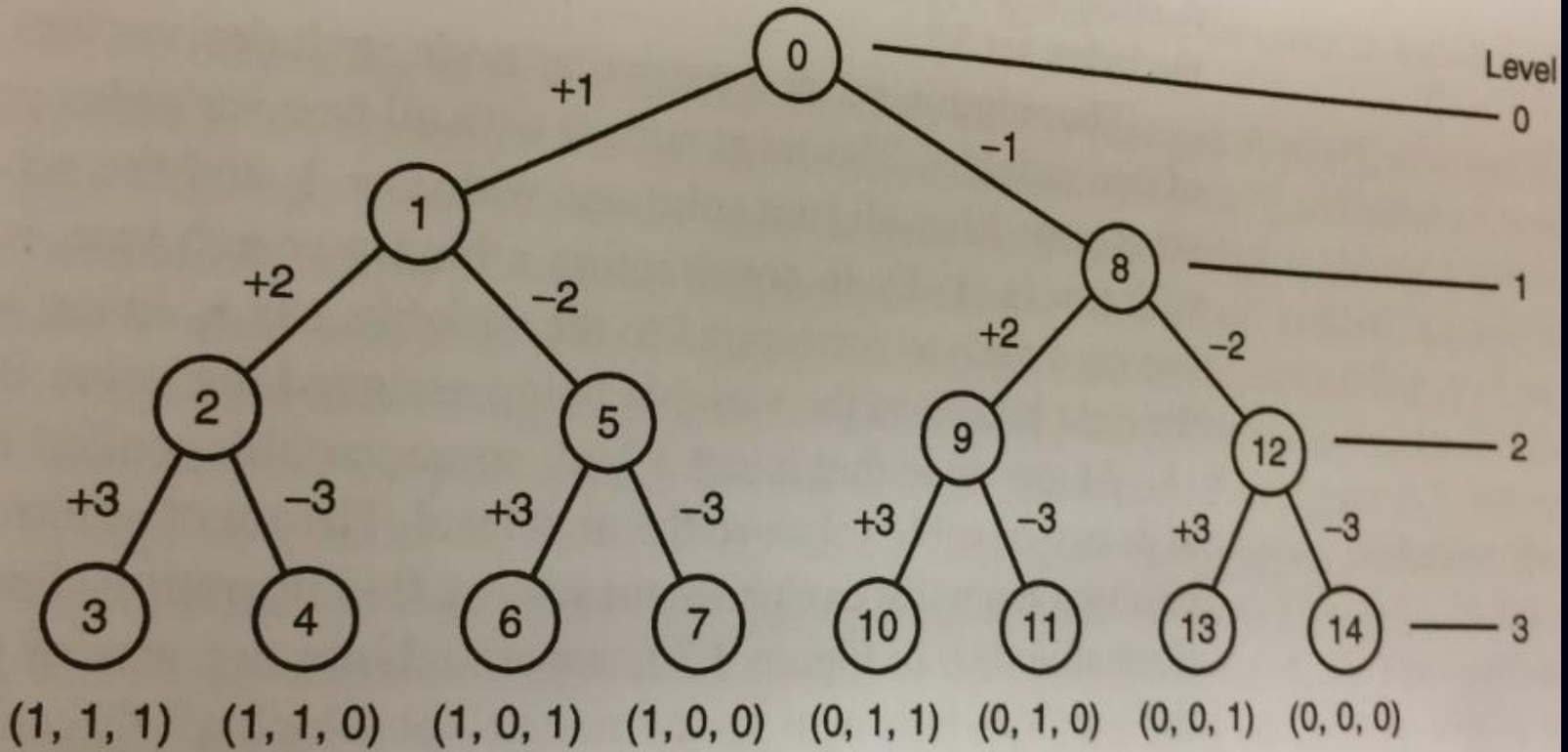


Figure 8. 3 Exhaustive search tree

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 xj has been set equal to 0, whereas a positive number, +j, implies that xj 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
- 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.

Nonlinear programming

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

Midterm on Tuesday 10/24

- Material will be from lectures (which obviously overlap a lot with the textbooks) and from homeworks.
- No programming or questions that require Python.
- No questions on material from the textbooks that was not covered in lecture.