CAP 4630
Artificial Intelligence

Instructor: Sam Ganzfried

sganzfri@cis.fiu.edu

HW2

- Due yesterday (10/18) at 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.
  - 5 total late days

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 (and parts of homework 2 if all students have turned it in by start of class), midterm review
- 10/24: Midterm
- 10/26: Planning

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

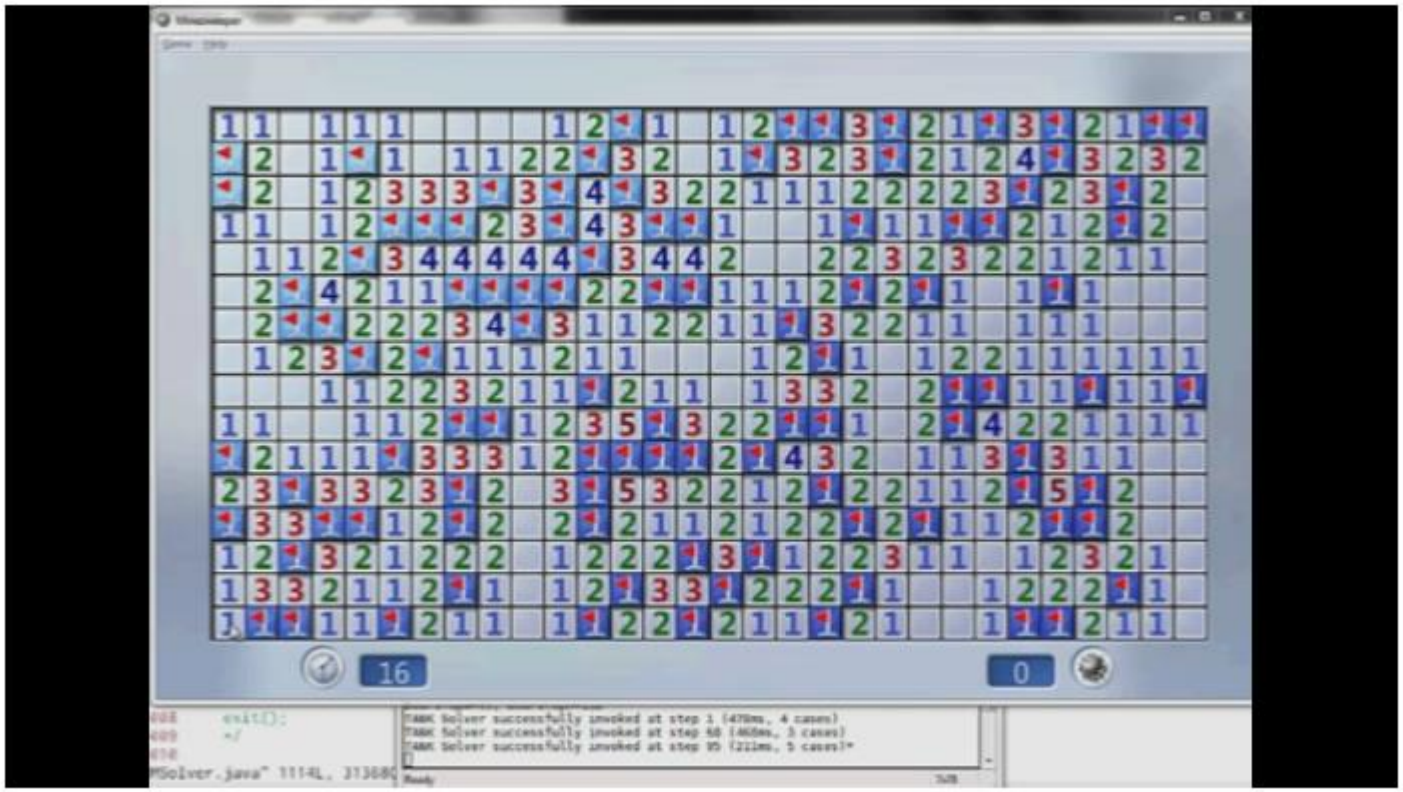
- P: polynomial-time algorithm exists
  - E.g., O(n), O(n^2), O(n^3), etc.
  - This is “efficient”
  - Most search algorithms we saw were NOT polynomial time
  - Many important AI problems can NOT be solved exactly in polynomial time
  - Theory does not always equal practice (e.g., poker, linear programming)
  - 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

- 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

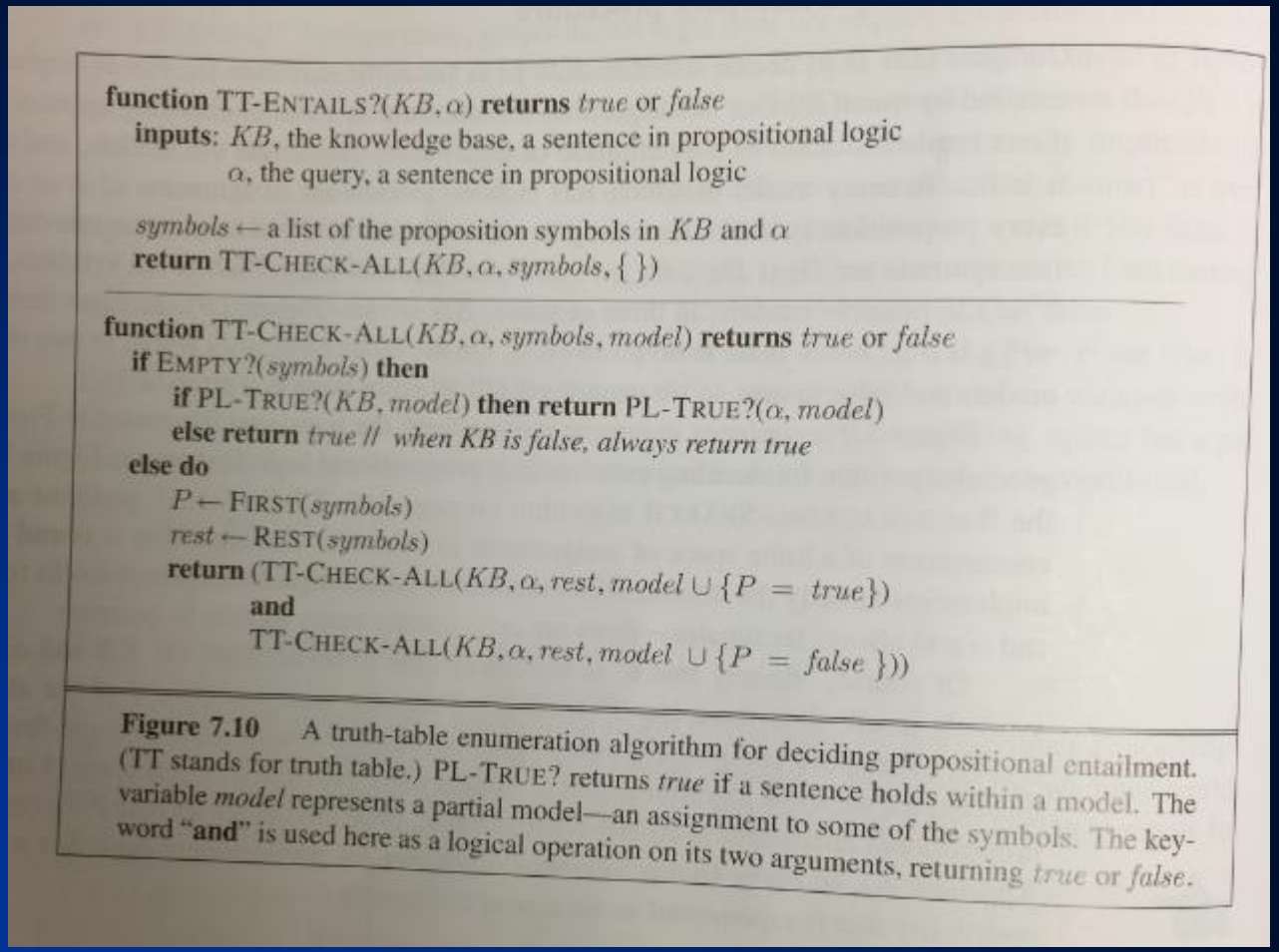
- Recall that a sentence A is **entailed** by sentence B if it follows logically from it using one of the rules we have seen.
- **Soundness** is highly desirable for logical inference: an unsound inference procedure "essentially makes things up as it goes along—it announces the discovery of nonexistent 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 = 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

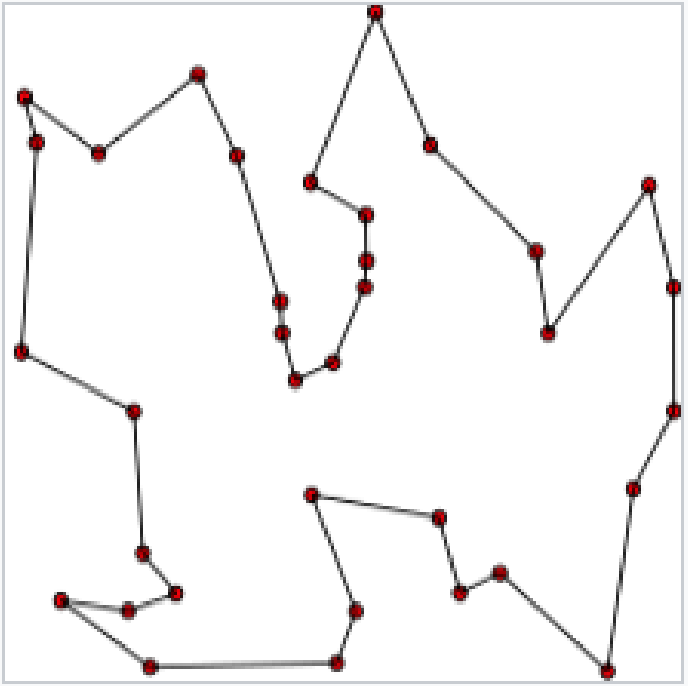
Manufacturing site selection

- 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:

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

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.
- 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

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

$$\text{subject to} \quad \sum_{j=1}^{n} a_{ij} x_j \le b_i, \quad i = 1, \ldots, m$$

$$x_j = 0 \text{ or } 1, \quad j = 1, \ldots, n$$

ILP search tree

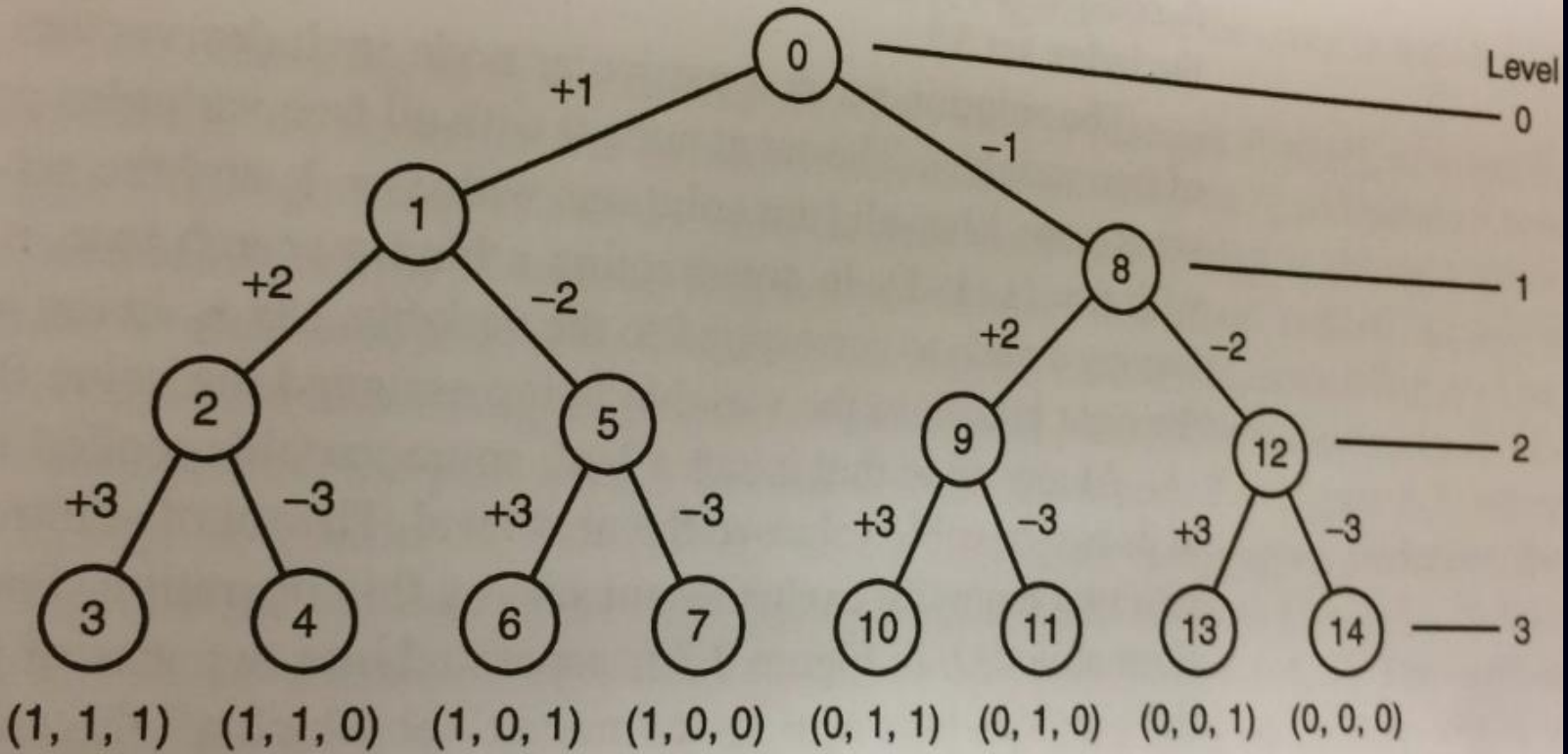


Figure 8. 3 Exhaustive search tree

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

Homework 1

- Solutions and graded assignments back today

Midterm on Tuesday 10/24

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

Midterm format

- ~30 Multiple choice questions
  - 3 pts each
- ~15 True/False with explanation
  - 4 pts each (1 point for True/False, 3 points for explanation)
- 5 Analytical exercises
  - 10 pts each

Midterm format

- All of the following search algorithms require an amount of space that is exponential in one of the problem parameters EXCEPT:
  a) BFS
  b) UCS
  c) DFS
  d) Bidirectional search

Midterm format

- True/False: In a search tree, BFS will always find a goal node with minimal depth if one exists?
  - If true, give a proof.
  - If false, provide a counterexample.
  - 3 points for correct true/false answer, 3 points for correct explanation.

Midterm topics

- Search
  - Uninformed search (e.g., BFS, UCS, DFS, DLS, IDS, bidirectional search, “Big four” search criteria)
  - Informed search (e.g., best-first search, A*, heuristic functions)
  - Local search (e.g., hill climbing, simulated annealing, genetic algorithms)
  - Adversarial search (e.g., minimax search, alpha-beta pruning)
  - Constraint satisfaction problems (e.g., inference vs. search, arc consistency, backtracking search, variable and value ordering)
- Logic
  - Propositional logic (e.g., wumpus world, models, truth tables)
  - Logical inference (e.g., entailment, model checking)
- Optimization
  - (Linear) integer optimization
  - Linear optimization
  - Nonlinear optimization