(http://csmatters.org) 5 - 5

0b101 - 0b101

Advanced Algorithms

Unit 5. Data Manipulation

Revision Date: Sep 19, 2019
Duration: 2 50-minute sessions

Lesson Summary

Summary: Students are introduced to the theory of computation, computability, the halting problem, and advanced algorithms. In particular, they will learn about heuristic search used by artificial intelligence (AI) programs to play games.

Objective:

Students will be able to:

- define computation and some basic ideas of the theory of computation
- discuss computability and understand there are some things computers cannot solve
- explain the Halting Problem
- identify some advanced search algorithms
- understand how AI programs represent games with game trees
- understand how AI programs use uninformed and heuristic search algorithms to play games

Overview:

Session 1

1. Getting Started (10 min)
2. Guided Activity (35 min)
    1. Inverse Operations Activity [10 min]
    2. Computation [10 min]
    3. Computability [15 min]
3. Wrap Up (5 min) - Think-Pair-Share

Session 2

1. Getting Started (5 min)
2. Guided Activity (45 min)
    1. Search and Game Trees [15 min]
    2. Game-Playing AI [10 min]
    3. Types of Heuristic Search [10 min]

4. Playing a Game with Heuristic Search [10 min]

Learning Objectives

CSP Objectives

- *EU AAP-2 - The way statements are sequenced and combined in a program determines the computed result. Programs incorporate iteration and selection constructs to represent repetition and make decisions to handle varied input values.*
    - LO AAP-2.L - Compare multiple algorithms to determine if they yield the same side effect or result.
- *EU AAP-4 - There exist problems that computers cannot solve, and even when a computer can solve a problem, it may not be able to do so in a reasonable amount of time.*
    - LO AAP-4.A - For determining the efficiency of an algorithm: a. Explain the difference between algorithms that run in reasonable time and those that do not. b. Identify situations where a heuristic solution may be more appropriate.
    - LO AAP-4.B - Explain the existence of undecidable problems in computer science.
- *EU CSN-2 - Parallel and distributed computing leverage multiple computers to more quickly solve complex problems or process large data sets.*
    - LO CSN-2.A - For sequential, parallel, and distributed computing: a. Compare problem solutions. b. Determine the efficiency of solutions.

Math Common Core Practice:

- MP2: Reason abstractly and quantitatively.
- MP4: Model with mathematics.

Common Core Math:

- N-RN.3: Use properties of rational and irrational numbers.
- F-BF.1-2: Build a function that models a relationship between two quantities
- F-BF.3-5: Build new functions from existing functions
- S-IC.1-2: Understand and evaluate random processes underlying statistical experiments
- S-CP.6-9: Use the rules of probability to compute probabilities of compound events in a uniform probability model
- S-MD.5-7: Use probability to evaluate outcomes of decisions

Common Core ELA:

- RST 12.3 - Precisely follow a complex multistep procedure

NGSS Practices:

- 2. Developing and using models
- 6. Constructing explanations (for science) and designing solutions (engineering)

Essential Questions

- How can computational models and simulations help generate new understanding and knowledge?
- What considerations and trade-offs arise in the computational manipulation of data?
- What kinds of problems are easy, what kinds are difficult, and what kinds are impossible to solve algorithmically?
- How are algorithms evaluated?

Teacher Resources

Student computer usage for this lesson is: **required**

Links to videos and online tools as indicated in the lesson plan.

- http (http://www.sorting-algorithms.com/)://www.sorting-algorithms.com/ (http://www.sorting-algorithms.com/) for sorting algorithm review
- The Halting Problem https:// (https://www.youtube.com/watch?v=92WHN-pAFCs)www.youtube.com/watch?v=92WHN-pAFCs (https://www.youtube.com/watch?v=92WHN-pAFCs) (7:52)

Alternative instruction could include the Towers of Hanoi problem and discuss the algorithm for solving it. Some demonstrations are available here:

- http://illuminations.nctm.org/Activity.aspx?id=4195 (http://illuminations.nctm.org/Activity.aspx?id=4195)
- http://www.mazeworks.com/hanoi/ (http://www.mazeworks.com/hanoi/)

Lesson Plan

Session 1

Getting Started (10 min)

Think-Pair-Share: In pairs, think about and try to answer each of the following questions:

- Given y = 7x + 4 and x=3 what are the steps to find y?
- Given y = 7x + 4 and y=3 what are the steps to find x?
- Factor 81,927,497 and 81,927,499. Can you figure out the steps?
- Multiply 431 x 433 x 439. What are the steps?

Note: just give them a few minutes to try the factoring, but round them up to continue and discuss: which operations were much harder to perform than their inverse? Can you just invert the steps, and why or why not?

Guided Activities (35 min)

Inverse Operations Activity [10 min]

1. Convey the following concepts:
    a. Inverse arithmetic operations
        - add/subtract [x + 7 – 7 = x – 7 + 7 = x] ;
        - multiply/divide [x * 7 / 7 = x / 7 * 7 = x] ;
        - e^x /ln(x) [$\ln(e^x) = e^{\ln(x)} = x$];
        - …
    b. Some arithmetic operations are harder to do then their inverse operations -- *as the students did during the warm-up*.
        - Cubing a number versus finding the cube root of the result.
            - Find the cube of 12
            - Find the cube root of 5832
        - Multiplying numbers to form a product versus factoring the product.
    c. The same can be true with algorithms.
        - It is much easier to scramble a Rubik’s cube with a few moves than it is to solve a scrambled Rubik’s cube with a few moves. *[Optional -- Have the students discover this using a Rubik's cube or an online simulated cube].*

Make a connection to the previous lesson by comparing these to sorting algorithms, where some are speedy and efficient like Merge sort and Quick sort, and others are unusably slow, like Bubble sort. Highlight the difference that different problems have different lower bounds on optimal solutions, and that some problems like integer factorization have solutions but take too long to be solved in a practical way.

Computation [10 min]

Discuss the definition of computation (in a theoretical sense) with your students. Computation is input plus processing to get output. A computer is one system that is a "model of computation" since it takes input, processes it, and produces output.

Another model of computation is called a Turing machine, named after Alan Turing (one of the most famous computer scientists). A Turing machine is a theoretical entity that has a tape of symbols (a line of 0s and 1s), a head that can read only one symbol at a time, and an internal state that can change based on instructions as the head reads symbols. Turing and a mathematician called Alonzo Church are responsible for the "Church-Turing" thesis, which says that a Turing machine can compute anything that a digital computer can. This is a fundamental idea of the theory of computation, and has the implication that anything one computer is capable of doing is possible to be computed by another, given enough resources (time and memory).

Parallel computing is a computational model where the program is broken into multiple smaller sequential computing operations, some of which are performed simultaneously. Distributed computing is a computational model in which multiple devices are used to run a program. Comparing efficiency of solutions can be done by comparing the time it takes them to perform the same task.

Computability [15 min]

Now discuss the idea of computability with your students. Ask your students to answer or think-pair-share: are there things it is impossible for a computer to compute? A *decision problem* that can be solved by an *algorithm* that halts on all inputs in a finite number of steps is decidable. The most classic "undecidable" (non-computable) question is called the Halting Problem. The Halting

Problem is: make a program that can tell if any other program will halt (terminate at some point eventually) or will loop forever and never end. This can be done for specific algorithms (instances of the Halting Problem) but not for the general problem regarding all possible algorithms.

The Halting Problem is impossible for a computer to compute, which you can prove (informally) by paradox. Suppose you *did* have a program that solved the Halting Problem, called HALT(X), which takes the code for some program X as input and says "yes" if X terminates or "no" if X loops forever. Then you could write a new program that uses HALT inside it, which we will call PARADOX(X). First PARADOX(X) will run HALT(X) and if the result is no, PARADOX will halt, but if the result is yes, then PARADOX will loop forever. But here is the problem: what if we use the code for PARADOX as the input to PARADOX, running PARADOX(PARADOX)? If it says that PARADOX halts, then PARADOX runs forever, and if it says PARADOX runs forever, then PARADOX halts. This problem *is* a *paradox* and does not make sense because the premise, that a program called HALT could exist, must be wrong! Therefore, the Halting Problem is impossible for a computer to solve.

Video explanation with optional student simulation

- Alonzo Church, an American, and Alan Turing, from the UK, independently proved in the 1930s – before computers actually existed – that there are some problems that computers will never be able to solve. View: The Halting Problem at: https:// (https://www.youtube.com/watch?v=92WHN-pAFCs)www.youtube.com/watch?v=92WHN-pAFCs (https://www.youtube.com/watch?v=92WHN-pAFCs) (https://www.youtube.com/watch?v=92WHN-pAFCs) [Optional 7:52] Have groups of students act out the machines in the video to determine whether they understand the basics of the proof.

Wrap Up (5 min)

Think-Pair-Share:

- Ask your students to think about the following algorithms, pair off, and reorder them based on worst-case computational complexity, with the fastest ones first and the slower (or undecidable) ones last:
  - Bubble Sort
  - Factoring large integers
  - Merge Sort
  - Binary Search
  - Taking attendance
  - The Halting Problem
- Discuss the orderings that a few groups came up with. Advanced groups could also try to guess the computational complexity:
  - Binary Search, O(log n)
  - Taking attendance, O(n) since you just read off the list in order
  - Merge Sort, O(n log n)
  - Bubble Sort, $O(n^2)$
  - Factoring large integers, $O(e^n)$, approximately exponential depending on the algorithm
  - The Halting Problem, undecidable

Session 2

Getting Started (5 min)

This session concerns advanced algorithms, in particular heuristic search, which is commonly used in artificial intelligence. Refresh your students' minds on the definitions of computation, computability, and undecidable problems. Additionally, mention the properties we consider when we compare algorithms:

- correctness
- ease of understanding
- elegance and style
- time/space efficiency

Guided Activity (45 min)

Search and Game Trees [15 min]

Introduce the idea of heuristic search, which is a class of algorithms used in many artificial intelligence programs. A heuristic is something that is used to find a good solution in a reasonable time, and a heuristic search algorithm is an algorithm that uses heuristics to determine how to search through some space.

A great way to introduce heuristic search is first to discuss game trees. A game tree is a structure that is used to represent the "space" of a game that an algorithm wants to search through.

Think of a game like chess: you make a move, the opponent makes a move, and the process continues until the ending conditions have been met (one player in checkmate or stalemate). A game tree is a mathematical structure used by AI and heuristic search algorithms to model the moves made in a chess game. At any turn, we can make a "tree" by drawing the root node as representing the current state of the board and drawing one branch under it for every possible move. In Tic-Tac-Toe, if you are the starting player, then the root node represents a blank board, and there will be nine branches, one for each possible move (each space where you could place your mark). Following a branch in the game tree takes you to a new node that represents the configuration of the game that results from having taken that move. In Tic-Tac-Toe, if I am the first player and place my X in the center space, I have "followed" that branch down the tree to a new node that represents the board with an X in the center space. The opponent then uses this node as the root of their game tree, and has a branch for each of their possible moves.

Think-Pair-Share: Have your students pair off and play a game of Tic-Tac-Toe and try to draw the game tree as they play it, drawing the nodes for each move they made and every potential branch from those nodes. Bring them back into discussion and ask them what if they had to draw out *every* node followed down *every* branch? Now ask them to imagine the game tree for chess, which has 20 possible moves on the first turn, 400 on the second, and many, many more as the game goes on. How can an artificially intelligent program learn to play chess when there are so many (too many) options? Chess actually has around 35^{100} nodes in its tree and 10^{40} legal states.

Game-Playing AI [10 min]

Heuristic search on game trees is one way AI programs are able to play games like chess. How good are computer game players?

- **Chess**: Deep Blue, a complex machine that used databases and heuristic search, beat Garry Kasparov (one of the best chess players ever) in 1997
  - Garry Kasparov vs. Deep Junior (Feb 2003): tie!
  - Kasparov vs. X3D Fritz (November 2003): tie!
  - http://www.cnn.com/2003/TECH/fun.games/11/19/kasparov.chess.ap/ (http://www.cnn.com/2003/TECH/fun.games/11/19/kasparov.chess.ap/)
- **Checkers**: Chinook is an AI program with a *very large* endgame database that is the world champion. Checkers, like Tic-Tac-Toe, is "solved," meaning there is a known optimal way to play the game to always win or force a tie. You can play a version of Chinook here:
  - https://webdocs.cs.ualberta.ca/~chinook/ (https://webdocs.cs.ualberta.ca/~chinook/)
- **Bridge**: "Expert-level" computer players exist (but no world champions yet).
- **Poker**: Computer team beat a human team, using statistical modeling and adaptation:
  - http://www.cs.ualberta.ca/~games/poker/man-machine/ (http://www.cs.ualberta.ca/~games/poker/man-machine/)
- Good places to learn more:
  - http://www.cs.ualberta.ca/~games/ (http://www.cs.ualberta.ca/~games/)
  - http://www.cs.unimass.nl/icga (http://www.cs.unimass.nl/icga)

Typically games modelled with game trees are 2-person games, players alternate moves, and they are zero-sum (meaning one player's loss is the other's gain). More complicated elements in such games may have include: hidden information (like other players' hands), chance (dice), or multiple players.

Playing a Game with Heuristic Search [10 min]

How does an AI program use heuristic search to play a game? Typically in these steps:

- consider all moves that are possible for the current turn
- compute what the new positions and configuration of the board (the "state") for each of those moves
- evaluate each state using some scoring function to determine which is better
  - for example, taking an opponent's piece is probably going to be evaluated more highly than simply moving a pawn
- make the move that results in the best evaluated state
- wait for your opponent to play, repeat

The key problems are:

- representing the state of the board
- generating the resulting states from every move
- evaluating the value of the resulting states

For evaluation, some function is typically coded or learned over time.

- For Tic-Tac-Toe, for board state n, an evaluation function could be:
  - f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you], where a 3-length is a complete row, column, or diagonal
- Alan Turing's function for chess with board state n:
  - f(n) = w(n)/b(n) where w(n) is the sum of the point value of white's pieces on the board at state n, and b(n) is the sum of black's pieces. The point values are those commonly used by professional chess players, pawn is 1 point, the queen is 9 points.

Types of Heuristic Search [10 min]

Refer to the "Advanced Algorithms" slides in the lesson resources folder for examples of uninformed search. For an activity, you may want to create a game tree for Tic-Tac-Toe and have your students walk through how each of the following algorithms would operate over it.

Uninformed Search are algorithms that work without a heuristic, using no information about the likely "direction" of the goal node. Algorithms include:

- depth-first search
    - starting at the root of the game tree, pick one branch and examine the node at the end of it, then pick one branch of that node and examine the even deeper node (hence, "depth-first") until you reach the end of the game, then go back one node, pick one of its branches, explore until you reach the end of the game, and so on
    - this search is unpractical for games with many moves that may go on indefinitely
- breadth-first search
    - starting at the root of the game tree, called A. It has n branches leading to child nodes each called B_1, B_2, and on to B_n, for some n nodes. Examine each of these children in order before moving onto the children of B_1, examining each of those in order, then examining the child nodes of B_2 then those of B_3, and so on.
    - this search is unpractical for games with many or variable numbers of moves per turn

For any games with variety and complexity, certainly for chess and even checkers, uninformed search is simply too slow because it is exhaustive. This problem is another example, like with sorting, where the efficiency of our solution matters a great deal. To get programs to play games, we need them to be efficient and intelligent about the number and quality of moves they consider.

Informed Search algorithms each follow some heuristic that uses information about the game to determine smart directions to explore. Examples include:

- best-first search ("greedy")
    - at every turn, take the branch leading to the node with the greatest value from the evaluation function
    - the problem here is when there is delayed reward since this "greedy" approach lacks any ability to look ahead. For instance, if there are two moves available, one that is great and another that is just decent, it will always pick the great move even if a winning move could be made the turn after the decent one.
- A* ("a star")
    - estimates the goal node and picks nodes based on the least cost. It follows a best-first strategy but also factors in the distance it has traveled from the original root node.
- Your GPS device! In order to figure out the best, fastest route to your destination, your GPS will search through the possible roads you can take intelligently by factoring in things such as the span and capacity of the road, the traffic, and potentially even the time of day.

Advanced classes may wish to discuss local search algorithms, such as hill-climbing and genetic algorithms (in the "Advanced Algorithms" slides in the lesson folder).

Minimax

Thinking about game trees again, we want to select the branch that takes us to a node with the maximum evaluated state. But there is a catch: the opponent gets to make moves, too. That is, every other branch in our game tree is the opponent's turn. How does the AI program account for the other player?

Perhaps most logically, the way AI programs do so is to assume the other player will play optimally. Just as the AI will take the branch that leads to the state with the greatest evaluation, it assumes the other player takes the branch leading to the state that will maximize *their* position. In other words, the AI searches through their game tree by following the branch with the maximum value on their turn, and following the branch with the minimum value on the opponent's turn. This algorithm is called minimax and is the basis of nearly all AI that play 2-person zero-sum games.

Options for Differentiated Instruction

For the Halting Problem proof, it is important that students can translate the solution that is on the video into a representation that makes (some) sense to them. Acting out the inputs and outputs of the set of machines is an approach worth trying.

Evidence of Learning

Formative Assessment

The following "Checks for Understanding" could be used to guide the students towards the three learning objectives.

Objective: Students will identify some Advanced Algorithms that Exploit Inverse Operations Efficiency.

1. Pairs of students will be asked to list pairs of basic arithmetic inverse functions.
2. The class will develop a composite list of inverse functions found by the student pairs. Note: many of these pairs share the same key on their graphing calculators.
3. Students will factor composite numbers and create the same composite numbers from their prime factorization. They will log the relative effort in their journals.

Objective: Students will identify some Advanced Algorithmic Techniques.

1. Students will find examples from earlier modules where the algorithms used techniques of heuristics, randomness, probability, etc. This could be a good group review of prior topics.

Objective: SWBAT discuss at least one example of a computing problems that is unsolvable

1. Students will either describe the proof of Turing's Halting problem using models in a way that is similar to that used in the lesson video or they wil build a similar physical model using students as the machines.

Summative Assessment

Students will be able to summarize -- in their own words or with simple models -- the proof of the Halting Problem.

Students will be able to identify the sensitivity of cryptography to the difficulty of factoring large numbers.

Authored by: CS Matters in Maryland
Website: csmatters.org (http://csmatters.org)
Email: csmattersinmaryland@gmail.com (mailto:csmattersinmaryland@gmail.com)