ECE 2195 – Homework 3
Avery Peiffer

1. Decision Diagrams, Decision Trees, Influence Diagrams

In general, decision diagrams and influence diagrams are compressed representations of decision situations. In the case of binary decision diagrams, a Boolean function is compressed into a form that offers faster retrieval than a traditional lookup table. Influence diagrams and decision diagrams are both directed acyclic graphs; however, influence diagrams seem to be a more generalized version of the decision diagrams that we have studied in class. Decision diagrams only have on type of node, while influence diagrams have multiple types of nodes with different representations. Decision nodes, drawn as a rectangle, correspond to the options when making a decision. Uncertainty nodes represent the information one possesses regarding a problem, and are drawn with an oval. Finally, value nodes represent utility and preference and are drawn as an octagon or diamond. The nodes are then connected with arcs to represent their relationships. Influence diagrams quantify the value of information, which represents how much a decision-maker is willing to give up to gain more information about a situation.

Decision trees are a data structure, heavily used in machine learning, that rely on segmenting the feature space into a number of regions using a set of splitting rules. A tree has a first rule at the root, followed by internal nodes that represent further splits. The leaves of the decision tree represent the different regions of the feature space. Decision trees can be useful for machine learning because they are easily interpretable; each node represents a different criterion that the model uses for splitting. Furthermore, nodes towards the top of the tree represent the more important rules. This allows users to understand the specific criteria upon which the tree is splitting. Additionally, decision trees can represent complex nonlinear models with a relatively low computation cost, especially compared to neural networks.

Like most machine learning techniques, decision trees are subjected to the bias-variance tradeoff. Very long trees have high variance and are prone to overfitting; the opposite is true for very small trees. To prevent overfitting, techniques can be used to prune the tree, limiting criteria such as the depth of the tree of the number of observations per region. Additionally, overfitting can be avoided by combining multiple decision trees together. These approaches are called random forests, bagging, and boosting. Combining trees can result in a higher accuracy, but also incurs a cost to the interpretability of the model.

Due to their interpretability and low overhead, decision trees have a wide range of applications in classification and regression settings. A common example of a decision tree is a system to automatically evaluate loan applications; such a system uses many features about applicants to determine whether they are a good candidate for a loan. While likely not entirely practical (since applications are likely to be evaluated more holistically), it is easy to see why decision trees could be useful in this application – some factors are more important than others when applying for a loan (e.g. credit score, income), and so would be found towards the top of the tree.

Sources:

- https://pubsonline.informs.org/doi/abs/10.1287/deca.1050.0047
- https://www.bayesfusion.com/influence-diagrams/
- Dr. Mai Abdelhakim's ECE 2195 lecture notes on decision trees

2. SAT solvers and applications

(i)

MiniSAT is an open-source SAT solver that is written in C and C++. The implementation uses several data structures and algorithms to implement the SAT solver. In the methods to see if a literal can be removed from a conflict clause, a stack is used to check all of the different clauses in an expression. Additionally, a heap is used to maintain the ordering of the variables. The code also uses garbage collection, similar to Java, to manage memory. All of these methods, when written in C/C++, allows the SAT solver to be more efficient than if written in a language with higher overhead, such as Python. MiniSAT is designed to be a minimalistic SAT solver, which is why its code seems to be fairly straightforward.

On the other hand, CryptoMiniSat is more complex, extending a SAT solver to cryptographic problems. This system is also built in C++/C, but contains a Python interface as well. The authors on the original paper adapted a SAT solver to be more compatible to cryptographic problems. Whereas ordinary SAT solvers work with the conjunctive normal form, this SAT solver was extended to understand the XOR operation, which is common in cryptography. The authors showed that CryptoMiniSat could solve several different cryptographic ciphers up to 64 times faster than normal regular SAT solvers. Again, it seems that C++ and C are used to actually implement the SAT solver, because they are much more suited for the necessary low-level operations.

Sources:

- https://github.com/niklasso/minisat
- https://link.springer.com/chapter/10.1007%2F978-3-642-02777-2_24

(ii)

Jussila, Sinz, and Biere used BDDs as steps in a proof in an extended resolution logical framework. They used this basis to create a SAT solver that proves if a formula is actually unsatisfiable. This is useful because modern solvers go to great lengths to reduce the search space of a problem, employing complex techniques to do so. If one of these techniques is inaccurately implemented, the solver may return incorrect results. In this work, intermediate BDDs are built to keep track of the ordering of each clause. Then, when it is found that an expression is unsatisfiable, an operation similar to Shannon expansion is used to recursively generate the proof of un-satisfiability. Overall, this lets users trust the results of the SAT solver, instead of worrying about possible false negatives.

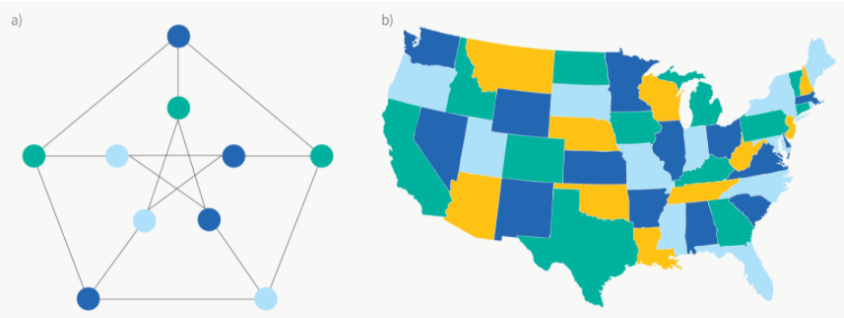
Bryant and Huele extended this work to perform existential quantification on any variable, as well as permit greater flexibility in the intermediate BDD ordering and the order in which operations are performed. Again, all of these operations are able to generate a proof that shows that the formula is unsatisfiable.

Sources:

- https://www.carstensinz.de/papers/SAT-2006.pdf
- https://www.cs.cmu.edu/~mheule/publications/tacas21-BDD.pdf

(iii)

One machine learning application of SAT solvers is in the graph coloring problem, which is a task to find the minimum number of colors such that no two adjacent vertices have the same color. Examples are shown in the figure below. The SAT expression is encoded based on the number of colors to test. Binary variables are then created, which are true if a vertex is colored with a given color. A constraint is then added that indicates that both members of the pair of variables cannot be simultaneously true. A SAT solver can then be run given these constraints, and the minimum number of colors needed can be found.



Another application of SAT solvers is in scheduling situations. Again, this can be represented by a set of constraints and Boolean variables such that all agents must be scheduled to rooms and times that fit them best. This representation can then be run by a SAT solver and a solution can be computed.

In machine learning specifically, the satisfiability problem can be used as function fitting, since many machine learning models fit complex nonlinear models to training data. SAT solvers can be used to represent a neural network by encoding the labels and output as Boolean variables, then setting the constraints for the network. A binary neural network was provided as an example on the Borealis AI website, in which the output label is the sign of the input multiplied by the model weights. The predicted label evaluates to true when more than half of the product terms evaluate to true, and false if less than half are true. For each training example, the model has one constraint, which is the provided label for that example. The SAT solver can then be run to see if there is a set of parameters for which all of the constraints are met. This example can then be easily extended to more complex multilayer networks.

SAT solvers can also be used to fit decision trees, finding the smallest possible tree that achieves a training classification performance of 100%. Boolean expressions are created that enforce constraints relating to a tree structure, such that all sets of variables that fit these constraints can form a valid tree. Several more Boolean expressions are then generated to represent that all the data must be correctly classified. When these two sets of constraints are ANDed together, it means that the SAT solver will return a tree that classifies all of the training data correctly. The user can then find the point at which the problem is unsatisfiable to return the most efficient tree that correctly separates the training data.

Sources:

- https://wiki.ubc.ca/images/f/f3/CSPSAT.pdf
- https://www.borealisai.com/en/blog/tutorial-9-sat-solvers-i-introduction-and-applications/

3. Controllability and observability

Controllability and observability are two relevant concepts in designing control systems. Controllability represents the ability for a dynamic system to go from any one state to any other state. If the system does not possess this ability, the system is uncontrollable. Similarly, observability represents the ability to uniquely determine the initial state of a system given the output and input.

Each of these concepts are represented by matrices that describe the system. Using the state equation of a linear system ($x' = Ax + Bu$), where x represents the state and u represents the input, the controllability matrix can be formulated as $C = [B\ AB\ A^2B\ \dots\ A^{n-1}B]$. N represents the number of time steps that are needed to travel to a particular target state. The controllability matrix has a solution if and only if the matrix has full rank, meaning that each of the rows are linearly independent. Notably, a matrix is full rank if and only if it has a nonzero determinant, providing an easy way to check if the system is controllable. The observability matrix can be formulated as $O = [C\ CA\ \dots\ CA^{n-1}]'$. Again, the system is observable if and only if the observability matrix is full rank, meaning it has a nonzero determinant.

These definitions from control theory make some sense when applied to the don't cares discussed in class. Controllability don't cares are associated with just the impossible patterns of inputs only, which is analogous to the emphasis on input and target states in controllable systems. Similarly, observability don't cares are refer to the output of a vertex in the network, which is again analogous to how observability depends on the output of a system.

An example of controllability and observability was provided by Dr. Zhi-Hong Mao, who described the Segway robot as a case study. The robot is modeled as a unicycle and inverted pendulum, which is highly nonlinear when written out as equations. When linearizing the equations around input and state = 0, the state equation is generated as shown below. Since A is 7x7, and B is 7x2, the controllability matrix will have 7 columns; however, its rank is only 6. This means that the linearized system is not controllable. Dr. Mao ascribed the issue to the linearization; linearizing around 0 causes the dynamics of the unicycle model to become unreliable. Therefore, a smaller representation of the system must be used in order for it to be controllable. In this case, that means removing the unicycle model of the system. Interestingly, this results in a controllability matrix with only 4 columns; this matrix ends up being full rank, which means that the system is controllable under this more simplified representation. The observability matrix follows a similar pattern for this problem.

$$\dot{\mathbf{x}} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.16 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 72.5 & 0 \end{bmatrix} \mathbf{x} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ -1.67 & -1.67 \\ 0 & 0 \\ 0.029 & -0.029 \\ 0 & 0 \\ -24.2 & -24.2 \end{bmatrix} \mathbf{u}$$

Sources:

- Dr. Zhi-Hong Mao's ECE 2570 lecture notes on controllability and observability