Chapter 2 Preliminaries

In this chapter we provide an overview of the mathematics and literature necessary to frame the results we present in the later chapters. We provide a survey of computational complexity and a discussion of flow network problems in Section 2.1. In Section 2.2 we provide details about computational social choice, voting rules, and a survey of bribery and manipulation in computational social choice.

2.1 Mathematical Background

In this section we give an overview of the different mathematical and computational constructs that we will use throughout the rest of this document. We review complexity theory in Section 2.1.1 and provide details of network flow problems in Section 2.1.2. This review is not intended to be a complete treatment of these ideas. Suggestions for further reading are included for each topic.

2.1.1 Computational Complexity

Much of the research presented in this dissertation deals with the computational complexity of reasoning in a variety of domains. We spend most of our time devising models that approximate some form of reality. Once we formally define these models we want to quantify how hard the problems are in a computational sense. In this section we hope to give the reader enough understanding about the area of complexity theory in order to follow our discussion in the later chapters. Complete treatments of these topics can be found in the books by Papadimitriou [99]; Garey and Johnson [65]; and Hemaspaandra and Ogihara [74]. This section is aimed at researchers from disciplines other than computer science and mathematics. In some cases we use less than precise explanations in order to give the reader a feel for the issues and we include mathematically precise definitions

for those who want to understand the material more exactly. Our hope is that we convey enough understanding of complexity theory for a non-technical reader to appreciate the technical discussion in later chapters. For these reasons, a reader familiar with complexity theory can safely skip this section.

Complexity theory seeks to quantify how hard a problem is in regards to a measurable quantity, typically, time or space. Time, in the sense of: "How many operations does it take to compute an answer?" Space, in the sense of: "How much memory is necessary to compute the answer?" These notions of time and space complexity are well formed ideas in the area of computational complexity and we give an overview here of what it means for a problem to be **hard**.

The first object that we must formalize is a computer. We will use a Turing Machine (TM) as our model of computation. A TM is a simple model of a computer. Even though a TM may seem like a toy model, it is powerful enough to capture the computational power of any modern computer [99, 120].

Definition 2.1.1 *A **Turing Machine (TM)** is a 4-tuple, $\langle Q, \Sigma, \delta, s_0 \rangle$, where:*

Q**:** *A finite set of states including s_0, s_{ACCEPT}, and s_{REJECT}.*

Σ**:** *The finite alphabet recognized by the TM. We assume that there is no difference between the input and tape alphabets. This alphabet must include $\sqcup$, the blank symbol.*

δ**:** *The transition function or program of the TM. δ is a mapping: $Q \times \Sigma \to Q \times \Sigma \times \{L, R, N\}$, where $\{L, R, N\}$ are Left, Right, and No movement of the tape head, respectively.*

A TM is a simple computer with a semi-infinite tape of symbols. These symbols are the alphabet used by the TM and must contain the "blank" symbol, $\sqcup$. This tape has a starting point on one end and is infinite in the other direction. One can imagine the computation process as involving a tape head which can read, write, and move on this tape (in either

direction) one symbol at a time. A TM starts its computation at the beginning of tape in state s_0 and reads a single symbol off of the tape. The TM uses its δ function, decides what symbol to write back to the tape and which direction to move the tape head (Left, Right, or No Movement). The TM, based on the symbols on the tape and the position of its tape head, moves through its computation, processing the symbols one at a time. Each symbol/movement pair describes a state, and every possible state is enumerated in the set Q. The TM will eventually reach s_{ACCEPT}, s_{REJECT}, or it will compute forever. A TM **halts** if it completes computation in either s_{ACCEPT} or s_{REJECT}. We say that a TM **accepts** a given input if the TM halts in s_{ACCEPT} and a TM **rejects** a given input if the TM halts in s_{REJECT}.

We use TMs to define *decision problems*. A decision problem is any problem that is answered with either a "yes" or a "no". For instance, the question, "Is the average of a set $U = \{u_1, \ldots, u_n\} \subset \mathbb{Z}^+ > t$?" is an example of a decision problem. We define decision problems in terms of free variables and relationships over those variables. The question, "Is the average of $U = \{3,4,5,6,7\} > 9$?" is an *instance* of the decision problem, "Is the average of a set $U = \{u_1, \ldots, u_n\} \subset \mathbb{Z}^+ > t$?" When we fix the free variables (the set U and t) to specific values ($\{3,4,5,6,7\}$ and 9) we create an instance of the decision problem. In this example the answer is "no" and we say the particular instance is not in the set of "yes" instances of our decision problem. Any specific decision problem (where we are given variables and relationships) defines a set of problem instances.

Definition 2.1.2 *Let Σ be a finite alphabet and Σ^* be the set of finite strings over alphabet Σ, including the empty string (ε). A **language** L over Σ is a subset of Σ^* ($L \subseteq \Sigma^*$).*

Definition 2.1.3 *If L is a language over Σ^* and M is a TM, then M* decides *L if and only if, for all $x \in \Sigma^*$, $M(x)$ halts, and $M(x)$ halts in s_{accept} if and only if $x \in L$ (else it halts in s_{reject}).*

Generally, we describe a decision problem as a specific language [65] and we create TMs to decide this specific language. We say that a given TM decides a language if it

accepts exactly those inputs that are in the language. Informally, we judge the power of our TMs by the types of languages they can decide given certain restrictions over the resources of the TM. We count how many operations it takes for the TM to decide instances of the language. We measure time complexity for a given TM as a function from the size of the input to the number of steps. This function gives us a relationship between the number of operations the TM must perform and the size of the input, e.g. the size of the encoding of the problem instance [99]. Note that we can construct an arbitrarily convoluted TM which takes many extra steps and, for this reason, we define the complexity of a language as the minimum complexity over all of its deciders.

We define **classes** of languages to be sets of languages with common properties. By a class, we mean a set of languages that can be decided with TMs that are constrained in similar ways (e.g. time complexity). By restricting the resources available to a TM, we gain an understanding of how much of some resource any computer would require to decide a language within a certain class. This idea provides a way to separate problems into classes based on how computationally hard they are to decide. The two most important classes we study are P, Definition 2.1.4, and NP, Definition 2.1.5 [74].

The difference between the classes P and NP is the notion of determinism. Informally, the class P is the set of languages that can be decided in time polynomial in the size of the encoding of the problem instance with a deterministic TM, while NP is the class of languages that can be decided in polynomial time with a nondeterministic TM. A nondeterministic TM is one that can select from a set of possible transitions at each computational step. It accepts an input if and only if there is a set of "good guesses" that allow it to reach s_{ACCEPT} when processing. Formally, in a nondeterministic TM the transition function δ becomes a relation instead of a mapping. This means that for any state and symbol pair there is zero, one, or many possible next states and the TM can choose any of these when performing a transition [99]. On the other hand, a deterministic TM is one that computes in the "normal way" most readers will be familiar with. By this we mean it computes one

thing after another and never guesses or deviates from its programming.

The resources we define our classes by are "TIME", in the sense of number of operations, and "SPACE", in the sense of number of bits of memory. In what follows we will use $f(n)$ to denote a "proper complexity function" as defined by Papadimitriou [99]. Formally, f is a function that maps $\mathbb{Z}^+ \to \mathbb{Z}^+$, a TM M that, for all x in $M(x)$, M takes exactly $f(|x|)$ units of some specified resource to complete its computation. Let DTIME(M) be the amount of time used by deterministic TM M to run an algorithm on a given instance given infinite space. Likewise, we let NTIME(M) be the amount of time used by nondeterministic TM M to run an algorithm on a given instance with no bounds on the amount of memory used [99]. We can similarly define DSPACE(M) and NSPACE(M) for the case of space resources. In the following definitions, we replace $f(n)$ with a specific family of functions parameterized by some integer $k > 0$. We define the complexity classes as the union of all the complexity functions using a certain amount of the specified resource.

Definition 2.1.4 *We define the class P as:*

$$P = \bigcup_k DTIME(n^k).$$

P is the class of languages L such that there is a polynomial time, deterministic Turing Machine that accepts L.

Definition 2.1.5 *We define the class NP as:*

$$NP = \bigcup_k NTIME(n^k).$$

NP is the class of languages L such that there is a polynomial time, nondeterministic Turing Machine that accepts L.

Definition 2.1.6 *We define the class EXP as:*

$$EXP = \bigcup_k DTIME(2^{n^k}).$$

EXP is the class of languages L such that there is a deterministic Turing Machine that accepts L in an exponential amount of time.

We can also define classes in terms of the amount of space they require.

Definition 2.1.7 *We define the class PSPACE as:*

$$PSPACE = \bigcup_k DSPACE(n^k).$$

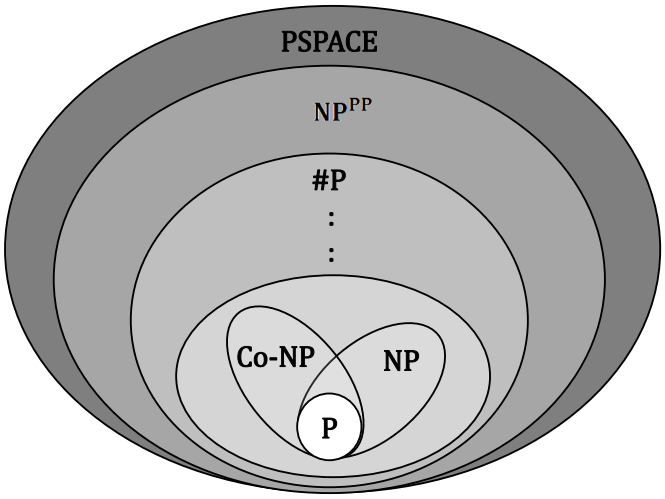
PSPACE is the class of languages L such that there is a polynomial space, deterministic Turing Machine that accepts L.

While we will not prove any theorems about EXP-time algorithms, they are a looming danger in several of the problems we study in later chapters. Many combinatoric problems may require brute force algorithms which take time exponential in the size of the problem input to compute an answer [100]. Any bribery scenario admits a simple algorithm: attempt every possible combination of bribes for every voter. However, this is not an efficient algorithm and would require EXP-time in the worst case. Our job is to figure out if there are better ways to compute the answers to questions about our model and avoid EXP-time algorithms.

Definition 2.1.8 *A language L is* $\in$ *coNP if and only if* $\overline{L} \in NP$.

The class of complements of NP languages is called coNP (Definition 2.1.8). For a given nondeterministic language L, determining if $x \in L$ requires only finding one accepting sequence of guesses (computation). However, if we want to show that $x \notin L$, we must check that no accepting sequence of guesses exists. This is in stark contrast to the class P, which is closed under complement. The question of whether or not nondeterministic TMs are closed under complement is one of the most important open questions in theoretical computer science [99].

Figure 2.1: An illustration of the ordering over the complexity classes. P is the computationally easiest class shown and PSPACE is the most computationally difficult class shown.



Generally, we say that a problem is **hard** if it falls into a class requiring more computational resources than problems in the class P. We can see the relationship among the complexity classes we use in this dissertation illustrated in Figure 2.1.1. We note that it is an open question as to whether P = NP. While the difference between P and NP seems obvious, it has not been directly proven that the two classes are different. It is also an open question as to whether or not P =?PSPACE. If P = PSPACE then the entire class hierarchy shown in Figure 2.1.1 would collapse (as all classes would be equal) [99].

In later chapters we consider several problems which may be in the function class #P introduced by Valiant [126]. Informally, #P is the counting analog of the class NP. For a given decision problem, rather than asking the question, "Does a solution exist?" the functional class #P asks, "How many solutions exist?"

Definition 2.1.9 *A function g is in #P if there is a nondeterministic TM M, such that $\forall x : g(x)$ is the number of accepting computations of $M(x)$.*

We can define a class of decision problems that are closely related to the functional class #P.

Definition 2.1.10 *A language L is in the class PP if there is a nondeterministic TM M such that, for all x, $x \in L$ if and only if $M(x)$ accepts on more than half of its computations.*

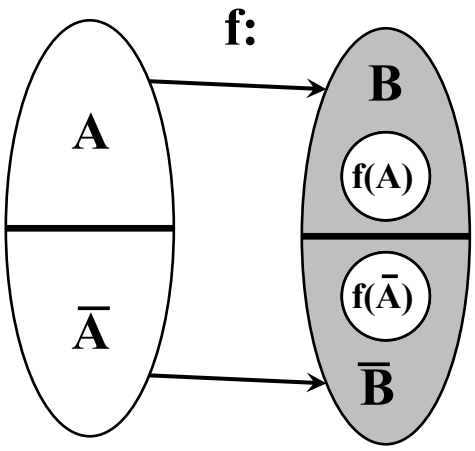
The class #P seems much harder than NP since a #P function requires that we count all the possible solutions to an NP-complete problem. We use Figure 2.1.1 to graphically illustrate the relationship between complexity classes. The class PP is the decision version of #P: we have to know all the possible accepting computation paths before we can decide if a majority of the computations end up accepting [99].

Above #P we include the complexity class $\mathrm{NP}^{\mathrm{PP}}$. This notation uses the concept of an **oracle**. An oracle A, in complexity, can be thought of as a unit cost sub-function. So, if we had sub-function A then the complexity class P^A is the set of TMs that decide in deterministic polynomial time with the use of A as an oracle. The class $\mathrm{NP}^{\mathrm{PP}}$ is the class of nondeterministic TMs that require access to a PP (or #P) oracle to check their work [74]. Additionally, we omit many (possibly infinitely many) classes in our diagram [99].

In order to classify a given problem we can leverage our knowledge of other problems that have already been classified. To make use of this knowledge we use reductions. A **reduction** is a technique whereby we use previously solved problems and algorithms to solve a new problem or use a known hard problem to prove hardness of a new problem. If we are given a problem, A, a reduction from A to B is a function, $f : A \to B$, that provides a transformation of an instance of A to an instance of B. This allows us to use algorithms for B in order to solve instances of problem A. This idea is illustrated in Figure 2.1.1. In this work we will focus on a specific type of reduction, the many-one, polynomial time transformation [74].

Definition 2.1.11 *Given two languages, L_1 and L_2, we say that $L_1 \leq_m^p L_2$, L_1 is **many-one, polynomial time reducible** to L_2, if there is a function f such that $x \in L_1$ if and only if $f(x) \in L_2$ and f is computable in polynomial time.*

Figure 2.2: An illustration of a reduction. Given an instance of problem A, we say f reduces A to B if all "yes" instances of A are transformed into "yes" instances of B and likewise for "no" instances.



Completeness of languages is the final property that we will use to understand our complexity hierarchy. Completeness is how we precisely place languages into classes.

Definition 2.1.12 *A language L is **hard** for a class C if and only if for all $L' \in C$, $L' \leq_m^p L$.*

Definition 2.1.13 *A language L is **complete** for a class C if and only if $L \in C$ and L is hard for C.*

Completeness (Definition 2.1.13) consists of both an upper and lower bound. Hardness (Definition 2.1.12) establishes a lower bound for completeness while inclusion defines an upper bound. Showing that a language L is hard for class C shows that it is at least as hard as any other language in C (up to a polynomial increase in complexity). This gives a lower bound on the complexity of L. We must establish both lower and upper bounds to establish completeness of a language [65]. The notion of completeness allows us to indicate the complexity of deciding a language not only directly but also in relation to other languages.

With the tools we have described in this section we can now classify problems into several complexity classes and determine if the problems are included in, hard for, or complete for some given complexity class. These tools will allow us to analyze our models of

voting, bribery, and manipulation, in terms of the difficulty of finding answers to decision problems.

Parameterized Complexity

We mention several results that relate to the theory of parameterized complexity. The field of parameterized complexity was introduced by Downey and Fellows as a complement to classical complexity theory [40]. The central notion of parameterized complexity is that one can determine what the effects of particular **parameters** are on the complexity of a problem. There is a very large difference between algorithmic running times of $\mathcal{O}(n^k)$ and $\mathcal{O}(2^k \cdot n)$, however, if k is small then the running times are closer together (asymptotically the same if $k = 1$) [62].

Parameterized complexity can also be expressed as hardness and completeness for levels of the $\mathrm{W}[t], t \geq 1$ hierarchy:

$$\mathrm{FPT} = \mathrm{W}[0] \subseteq \mathrm{W}[1] \subseteq \mathrm{W}[2] \subseteq \cdots.$$

Languages in FPT are also in P (for a fixed value of the parameter). In some of our later results we provide reductions from a problem that is known to be $\mathrm{W}[2]$-hard [62]. The class $\mathrm{W}[2]$ contains what are referred to as the "hardest" NP-complete problems that, in general, are not closely approximable such as k-DOMINATING SET [62] There is also a special notion of parameterized reductions which varies slightly from classical complexity theoretic reductions. We refer the reader to [62] for a more complete treatment of the topic.

2.1.2 Flow Networks

Several of our proofs in later chapters will make use of algorithms for flow networks. Flow networks are extremely important in many areas and are used for a variety of problems including maximum bipartite graph matching [35], modeling shipping networks, and modeling the flow of electricity in circuits [1]. The study of flow networks was introduced by

Ford and Fulkerson [64] and is important in computer science as evidenced by the chapter long treatment in the book by Corman et al. [35], one of the standard introductory algorithm textbooks in computer science.

Let $G = (V, E)$ be a graph. A graph contains a set of nodes or vertices V and a set of edges E connecting the vertices. Let $e = (u, v)$ and $e' = (v, u)$. In an undirected graph e and e' are the same. Informally, edges on a directed graph can only be traversed in one direction while in an undirected graph the edges are two-way streets.

A flow network is a directed, acyclic graph modeling a system in which we want to move some amount of resource from a source to a destination (sink). Each edge in the network has a finite capacity and a finite cost associated with it and we want to find an assignment of flow to edges such that certain constraints are not violated. Flow networks and the questions about them come in a variety of styles and flavors and each has a wide array of algorithms associated with them. We will not survey all the methods and flavors here; we point the reader to the book by Ahuja et al. [1] for a more complete discussion.

In this dissertation we will only be using the MINIMUM COST FEASIBLE FLOW problem.

Name: MINIMUM COST FEASIBLE FLOW

Given: A flow network $G(V, E)$ which is an undirected graph with source $s \in V$ and sink $t \in V$ where edge $(u, v) \in E$ has capacity $c(u, v)$, flow $f(u, v)$, cost $a(u, v)$, and minimum flow $d(u, v)$. The cost of sending flow across an edge is $f(u, v) \cdot a(u, v)$.

Question: Minimize the total cost of the flow: $\Sigma_{u,v \in V} f(u, v) \cdot a(u, v)$ under the constraints:

Capacity constraints: $\forall (u, v) : f(u, v) \leq c(u, v)$

Required flow: $\forall (u, v) : f(u, v) \geq d(u, v)$

Skew symmetry: $\forall (u, v) : f(u, v) = -f(u, v)$

Flow conservation: $\Sigma_{w \in V} f(u, w) = 0$ for all $u \neq s, t$.

In general, a flow is a function $f : V \times V \to \mathbb{R}$ and the MINIMUM COST FEASIBLE FLOW problem has a polynomial time algorithm. However, if we restrict our costs, capacities, and minimum flow constraints to be integers, the MINIMUM COST FEASIBLE FLOW will have an integer solution, if any solution exists, and this solution is computable in polynomial time [1]. This is good news for us, as we will see in later chapters, as we can model certain forms of bribery as flow problems and the guarantee of an integer solution provides us with a powerful tool.

2.2 Social Choice and Preference Aggregation

For thousands of years societies have not only had to work together but also make decisions together. The field of social choice is largely considered to be first established by the Marquis de Condorcet [27, 106] with a book published in 1785. In this work, Condorcet begins to carefully outline what makes good voting procedures and what properties are important to individuals who have to vote. This principled approach laid the ground work for the study of social choice. In this section we survey voting rules, their properties and how computer science interacts with voting and social choice.

Computational Social Choice (ComSoc) is still a relatively young field in computer science. In this section we attempt to provide the reader with an understanding of the basic concepts in the area and references to relevant literature. A more focused literature review accompanies each chapter of this dissertation. There have already been many great dissertations in ComSoc, and many of these provide a good introduction to the major research in the field related to voting and preference aggregation: Conitzer [28], Faliszewski [50], Pini [101], Procaccia [104], Uckleman [125] and Xia [134]. There are also some excellent review articles over specific sub areas of ComSoc including the current state of bribery and manipulation by Faliszewski [59], an overview of all the major areas of ComSoc by Chevaleyre et al. [23], and an introduction to existing work in ComSoc as it relates to the use of CP-nets by Conitzer et al. [20].

ComSoc and social choice are related by two main bridges: taking results from social choice and using them in computational systems (for decision making; recommendation; coordination and control of multi-agent systems; and other uses [119]) and looking at classical social choice research from a computational point of view (the complexity of determining winners in voting rules, computational properties of voting rules, computational aspects of other algorithms [23]). This two-way exchange of information has resulted in an explosion of scholarship both in the computer science and social choice communities. While we focu sspecifically on voting in this dissertation we mention that other major topics in ComSoc are cake cutting and fair division algorithms; coalition formation; resource allocation; belief merging; and judgment aggregation [23]. The unifying factor in all these areas is that we are asking a group of agents (human or automated) to work together in some meaningful way.

2.2.1 Voting and Common Voting Rules

The variety of voting rules and election models that have been implemented or "improved" over time is astounding. For a comprehensive history and survey of voting rules see Nurmi [97], Tideman [123], Taylor [122], or Arrow et al. [3]. Any of these great references provides a more complete survey of the history, development, and axiomatic properties of voting rules. In this section we discuss several voting rules that we will use in later chapters, some axiomatic properties that are considered with respect to voting rules, and a brief remark about tie-breaking in voting rules.

Throughout this dissertation we use several notation conventions. We refer to the set of candidates as C with size m; the set of voters is V of size n. We assume generally, and note specifically in the following chapters, that the set of voters is represented in some reasonable way by their preferences. This can either be a strict linear order, a CP-net, or some other way of compactly representing how voter v_i feels about the candidates in C. We are also provided a voting rule E that will aggregate the preferences of the voters and return

a winner (or a set of winners) from the set C. Occasionally, we will require the definition of a **social welfare function** [122].

Definition 2.2.1 *Given a set of candidates C and a set V of preferences over the elements of C, a **voting rule** (or social choice function) E returns a non-empty subset of C.*

Definition 2.2.2 *Given a set of candidates C and a set V of preferences over the elements of C, a **social welfare function** W returns a linear ordering over the elements of C.*[1]

We now survey some of the more common voting rules.

Positional Scoring Rules: The family of voting protocols that fall under the framework of positional scoring rules include plurality, veto, k-approval, and Borda (among others). In these methods, a scoring vector $\vec{S}$ of length m is associated with the rule. Each entry in $\vec{S}$ is an integer and $\vec{S}$ is non-increasing. Each voter ranks the candidates in C in some position within this vector. We then, for each candidate, sum the points assigned to that candidate by all voters. The winner of the election is the candidate who has the highest total score. For a multi-winner election, the set of candidates who tie with the highest score are the winners.

Plurality: Plurality is the most widely used voting rule [97] and, to many Americans, synonymous with the term "voting"). The Plurality score of a candidate is the sum of all the first place votes for that candidate. No other candidates in the vote are considered besides the first place vote. Thus, the scoring vector is $\vec{S} = [1,0,0,\ldots,0]$. The winner is the candidate with the highest score.

Veto: Veto is often referred to as anti-plurality. In a veto election, all candidates but the last placed candidate receive a point. Thus, the scoring vector is $\vec{S} = [1,1,1,\ldots,0]$. The winner(s) is the candidate with the highest score.

[1] This is sometimes called a strict social welfare function [122].

k-Approval: Under k-Approval voting, when a voter casts a vote, the first k candidates each receive a point. In a 2-Approval scheme, for example, the first 2 candidates of every voter's preference order receive a point. Thus, for some fixed k, the scoring vector is $\vec{S} = [1, 1, \ldots, 1_k, 0, \ldots, 0]$. The winner of a k-Approval election is the candidate with the highest total score.

Borda: Borda's System of Marks involves assigning a numerical score to each position. In most implementations [97] the first place candidate receives $m-1$ points, with each candidate later in the ranking receiving 1 less point down to 0 points for the last ranked candidate. Thus the scoring vector is $\vec{S} = [m-1, m-2, \ldots, m-(m-1), 0]$. The winner is the candidate with the highest total score.

Condorcet's Rule: While not technically a voting rule, the Marquis de Condorcet proposed that the winner of an election should be the alternative that is preferred, pairwise, to all other alternatives [27, 97]. This method, called the Condorcet Method, compares all pairs of alternatives and selects the one that wins, by majority, in *all* pairwise elections. This method does not necessarily have a winner.

Copeland: In a Copeland election each pairwise contest between candidates is considered. If candidate a defeats candidate b in a head-to-head comparison of first place votes then candidate a receives 1 point; a loss is -1 and a tie is worth 0 points. The particular number of points assigned can vary depending on the particular implementation and many different versions exist [3]. After all head-to-head comparisons are considered, the candidate with the highest total score is the winner of the election.

Voting Trees: Here we use the term "tree" fairly liberally: we define a voting tree to be any voting system that includes an ordered set of comparisons between the candidates. These voting trees can be in the form of a complete binary tree (the cup rule) or as a linear system of comparisons (linear balloting). The tree structure defines the order of comparisons between the candidates when each candidate is assigned to

a leaf node of the tree structure. We perform a majority comparison between the candidates, promoting the winner, until we have a single candidate at the top of the tree.

Repeated Alternative Vote: Repeated Alternative Vote (RAV) is an extension of the Alternative Vote (AV) [97] into a rule which returns a complete order over all the candidates [61]. For the selection of a single candidate there is no difference between RAV and AV. Scores are computed for each candidate as in Plurality. If no candidate has a strict majority of the votes the candidate receiving the fewest first place votes is dropped from all ballots and the votes are re-counted. If any candidate now has a strict majority, they are the winner. This process is repeated up to $m-1$ times [61]. In RAV this procedure is repeated, removing the winning candidate from all votes in the election after they have won, until no candidates remain. The order in which the winning candidates were removed is the total ordering of all the candidates.

With the selection of a voting rule it is almost as important to consider the selection of the tie-breaking method. Many times, candidates will tie with the same number of votes but we require a single winner. Oftentimes, in this dissertation, we speak of voting rules as returning a winning set of candidates and thus, we do not require a tie-breaking method. However, sometimes we will need a single or unique winner. In these cases we will define the tie-breaking method that we employ for the particular problem under consideration. There are a variety of tie-breaking methods used in the literature including lexicographic linearization, randomized tie-breaking, partial re-voting, and, in the state of New Mexico, "any reasonable game of chance such as poker or craps [77]." The effect of particular tie-breaking methods on the reasoning complexity of problems studied in later chapters can be significant and, often, tie-breaking is a completely separate line of investigation when studying voting rules [58, 98].

Axiomatic Properties of Voting Rules

Axiomatic characterizations of voting rules is one way that social choice theorists have, over the years, attempted to answer the question, "which voting rule is the best?" We will only scratch the surface of the field in this section. The references in the last section provide a more complete characterization of voting rule properties. Some properties that will be important to us are the following:

Condorcet Criterion: A voting rule that always returns the Condorcet winner, when one exists, is said to obey the Condorcet criteria or be **Condorcet consistent**.

Majority Criterion: The majority criteria is a slightly weaker version of the Condorcet criteria. A voting rule obeys the majority criteria when it always selects the alternative that wins a majority of its head-to-head comparisons by a majority vote.

Resoluteness: A voting rule is resolute if it always picks exactly one winner out of the set of alternatives. Similarly, a social welfare function is resolute if it returns a linear ordering of all the alternatives.

Non-Dictatorship: A voting rule is non-dictatorial if there is no voter v_i such that the result of the election always matches the preferences of v_i. This means that the voting rule considers all voters and does not just return the preference profile of some special voter.

Pareto Optimality: A voting rule is Pareto optimal if, for all pairs of alternatives x and y, if all voters prefer $x > y$ in their individual preferences, then the result of the voting rule will have $x > y$.

Independence of Irrelevant Alternatives (IIA): A rule satisfies IIA if, for any pair of alternatives x and y, if x ranked ahead of y in the social welfare ordering and some set of voters changes their votes, but no voter changes the relative ordering of x and y in their preference list, x should still win the election. For voting rules, we modify

the definition slightly to say that if x is in the winner set, then moving y without interchanging the relative ordering x and y should not remove x from the winner set.

These axioms, along with many others defined in the social choice literature, provide us some means of talking about voting rules with respect to the properties of individual voting rules. We provide additional discussion about the evaluation of voting rules and the study of the particular issues related to certain voting rules in Chapter 5. The properties we have discussed so far do not tell us anything about the security of voting rules against attacks of various forms, which is the focus of this dissertation.

2.2.2 Affecting Elections: Bribery, Manipulation and Control

Making group decisions is hard, and a primary concern is the fairness of these group decisions. One aspect that concerns many voters is the fairness of the process: did someone or something get chosen as a winner that should not have been? Did this particular candidate win the election through some form of cheating or manipulation? These questions form the basis for evaluating the safety and security of voting rules.

The field of preference aggregation manipulation originally stems from the field of social choice. The cornerstone text, Arrow's *Social Choice and Individual Values*, shows that any preference aggregation scheme for more than three alternatives cannot simultaneously satisfy Pareto optimality, non-dictatorship, and be independent of irrelevant alternatives [2]. Arrow's principled look at choice procedures led to a cascade of work in the field of social choice and raised new questions about the fairness of voting rules. Building on Arrow's work, the Gibbard–Satterthwaite Theorem shows that any aggregation system, meeting the same set of properties as Arrow's Theorem, can be manipulated by non-truthful voting [67, 115]. The Duggan–Schwartz Theorem extends the Gibbard-Satterthwaite Theorem to an even larger set of aggregation methods by removing the requirement of resoluteness [41].

Taking the results of Arrow and Duggan–Schwartz, we are left with the result that we cannot devise a fair and resolute preference aggregation scheme that is immune to manipulation. This seems unsettling on many levels because it implies that groups can never come to fair, non-manipulated agreements. However, in the late 1980's and early 1990's Bartholdi et al. proposed the idea of protecting the aggregation schemes through computational complexity [5, 7]. The idea, much like the central idea of cryptography, is that if it is difficult to compute a manipulation scheme then it is unlikely that there will be manipulation. With this idea of security in mind, much of the work in the ComSoc community seeks to classify aggregation systems in terms of their susceptibility to manipulation.

There is an ongoing discussion in the ComSoc literature about the quality of the protection afforded by computational complexity [32, 105]. In many real world cases, the number of candidates or voters is extremely limited. In these cases, even NP-completeness is not enough protection since it only tells us about the worst case, not the average case. However, in cases where we have a very large set of alternatives NP-completeness may be enough. The ComSoc community is actively investigating the sufficiency of NP-completeness as a computational barrier for protecting elections and enforcing honesty by the participants [56].

In the study of election systems we generally talk about three different ways to affect elections and social choice systems: bribery (influence), manipulation, and control. In bribery, each voter has a preference over the possible outcomes; we ask, can an outside actor change individual agents' votes in order to make some preferred outcome a winner? In manipulation, we are given a set of voters among all the possible voters; we ask, can we change the votes of only the given set of voters to make a preferred outcome a winner? In control, we are given all voters and their votes; we ask, can we change the requirements of the election (through adding or deleting voters or outcomes; or by changing the order of the comparisons in a voting tree) in order to make a preferred outcome a winner?

From this framework of ideas there stems a large collection of literature on the compu-

tational complexity of affecting elections. A survey by Faliszewski et al. gives a strong overview of the current work in the area of bribery and control [57]. There has also been work on the worst-case complexity of manipulation of voting mechanisms by voters [6, 30, 33] and average case complexity of manipulations [32, 47]. We provide a more focused look at the literature on bribery and manipulation as they relate to our models in Chapter 3 and Chapter 4.

In this dissertation we primarily focus on the bribery and manipulation problems. There has been a volume of research in computer science that address bribery and lobbying in deterministic domains [24, 49, 50, 52]. However, the study of manipulation is not the sole purview of computer science. There is a rich overlap between computer science and other disciplines including applied mathematics, operations research, and game theory. The search for a "better" preference aggregation function is an ongoing area of research. See the books by Taylor and Arrow for a current overview [3, 122].

Game theory [84, 127] has made, and continues to make, significant impacts on the study of voting and lobbying behavior. Shapley and Shubik studied the power of members in a committee system with voting represented as a simple game [118]. Operations and economic researchers have studied the lobbying process for both the US and European systems [36, 110]. One of the earliest formal models we can find of the lobbying process, as it is performed in the United States, is an operations research paper by Reinganum [110]. Baye et al. use the economic idea of "rent seeking" to study the lobbying process [8, 9]. While these studies focused only on the game theoretic aspects of behavior equilibriums, we focus on the complexity of finding computationally efficient manipulation schemes.

We also mention the increasing overlap between the economics research and the computer science research. It is important to understand that these disciplines pursue their research ends for different reasons but that significant overlap and cross-pollination exists. A good example of these inter-related ideas is a paper by Sandholm et al. in which auction generalizations are used to manipulate robot agents [114]. In addition, the book by Nisan

et al. provides an overview of game-theoretic aspects in use within multi-agent artificial intelligence systems [96].

Politics, Bribery and Real Life

There has been a move in the ComSoc community to reframe the discussion of the bribery problem in a more positive light. Initially the idea was to investigate the robustness of voting rules to various attacks [52]. However, the bribery problem is really a problem of resource allocation: any resource that can be distributed unevenly among the entrants can be used to affect the result. These resources could be referees or home fields in sports; volunteers and canvassers in political elections; money spent on targeted advertising and product placement; concessions made to friends; or subsidies in a spending bill. The ComSoc community uses the term "bribery" when unequal distribution of resources is a more general viewpoint. Due to recent developments on the bribery problem presented by Schlotter et al. [116], Elkind and Faliszewski [42], and some in this dissertation and supporting publications, the ComSoc community has started to look at these problems in a more positive light.

In this dissertation we focus on bribery and manipulation of preference aggregation functions. Arguably, the most important use of these functions in modern life are elections. Elections attempt to aggregate societal preferences (about politicians or pop songs) into winners and losers. The winners of these elections become presidents or albums of the year. Some elections are more important than others, and here we consider how these problems relate to political elections. Of course, when we speak of manipulation and bribery we are not advocating these procedures, however, it is important that we understand how susceptible our current procedures are to non-truthful and manipulative practices so that we can better secure them against these malicious actions.

Political scientists study voting behavior of congressional members under many different lights. Poole and Rosenthal provide a review and models for modern spatial voting

theory [102]. This method classifies representatives in a two-dimensional voting space and uses this classification to predict and understand voting behavior. Thanks to computers and the Internet it is now possible to perform powerful statistical analysis on both roll call voting data [133] and political contributions [63]. There is a rich literature which attempts to address issues of representation and influence in the US political system. While the literature does not show a direct correlation between moneyed contributions and roll call votes, it allows for the conclusion that all constituents are not necessarily represented equally [69, 83]. We focus our discussion on determining what factors exert the most influence on these decision makers and how these factors can contribute to our models of influence and manipulation.

We cannot conclusively show that money directly buys roll call votes. In a paper by Hall and Wayman money is tied to a "participation metric" [69]. The model developed by the authors assumes that money buys access to a representative. The author tracks contributions to see what effects the contributions have, if any, on the behavior of the representatives. The conclusion is that time and information are the most important resources to a congressional member. Therefore, money and resources have some effect on congressional behavior but not necessarily a direct effect [69].

The findings by Hall and Wayman are reinforced in a paper by Clinton which develops a model to determine who is represented by congressional members [26]. Clinton formulates novel metrics to detail how responsive a congressional member is to their constituents. Clinton uses local survey data from constituents to determine a district's true voting preference and compares this with observable representative votes. He concludes that constituent preferences do not (necessarily) determine how the representative votes. This conclusion holds both for a large set of 800 roll call votes (within one congress) and 25 "key votes" on issues the author identifies as "highly salient", including health care, war, and abortion. The paper shows large systematic differences between what constituents desire and how representatives vote. Clinton speculates that the discrepancy could be due to "party influ-

ence, presidential influence, or other outside actors [26]." The conclusion that there are hidden influence structures within the US congressional system is supported by a different model developed by Levitt [83]. Clinton also studies voting patterns using Bayesian models for estimation and inference of congressional members' ideologies to account for these ideological discrepancies [25].

An important distinction to keep in mind is that not all votes in the US Congress are roll call votes. Other types of votes include quorum calls, committee votes, and subcommittee votes [102]. Therefore, looking at just roll call votes may not be enough to determine all the influence factors that come into a representatives' decision. In fact, most of the content of bills is written by committees long before a bill ever goes up for a general vote. Most lobbying occurs during this process and we must temper our expectations of how much we can learn from an analysis of roll call data [48].

We use the political science research as a tool to inform our models where appropriate. By looking at others' research to determine what influence factors come in to play during voting decisions, we hope to better construct our models. The complexity of the particular voting aggregation method is only one factor which goes into manipulation. It is important to see if, given varying influence factors, the voting procedures retain their easy manipulation results.