CSE 421
Introduction to Algorithms

Lecture 28: Dealing with NP-completeness:
Fixed Parameter Tractability
SAT Solving
Reminder/Announcement

• The Final Exam is Monday December 11, 2:30-4:20 pm here
  • If nobody has a conflict that would prevent them staying longer, I will extend
    the time available to 4:45 pm.
  • Email me (enter a Private post on Edstem) by the end of day today if you
    can’t stay that long.

• See the pinned Edstem posts on Final Exam Information.

• I will run a Zoom review session on Sunday. Fill out the Edstem poll
  about the time for this session by end-of-day today.
Fixed Parameter Algorithms

The theory of fixed parameter tractability looks at NP problems using a second parameter $k$ in addition to input size $n$ and seeks algorithms with running times $f(k) \cdot n^{O(1)}$ where $f$ might be exponential.

**Clique:** Extra parameter $k$ for clique size target:
Brute force algorithm: try all subsets of size $k$ and check: $\Theta(k^2 n^k)$ time.

**Vertex-Cover:** Extra parameter $k$ for clique size target:
Brute force algorithm: try all subsets of size $k$ and check: $\Theta(mn^k)$ time.

• Neither is a good fixed parameter algorithm
Vertex-Cover Fixed Parameter Algorithm

Vertex-Cover$(C, b)$ {
    if there is an edge $(u, v)$ not covered by $C$ {
        if $b > 0$ {
            Vertex-Cover$(C \cup \{u\}, b - 1)$
            Vertex-Cover$(C \cup \{v\}, b - 1)$
        }
    } else
        Output YES (and set $C$) and halt
}

Call Vertex-Cover$(\emptyset, k)$
if no answer, output NO

Analysis:

- Time to identify possible edge $(u, v)$ not covered (and modify $C$) is $O(m + n)$
- # of recursive calls $\leq 2^{k \cdot n}$
- Total runtime $O(2^{k(m + n)})$
More on Fixed Parameter Algorithms

Many graph problems can be given a second parameter $k$ called the \textit{treewidth} of the input graph.

- Treewidth 1 graphs are trees (technically forests).
- Multiple natural definitions of treewidth (here’s one):
  - Graph $G = (V, E)$ is treewidth at most $k$ iff there is a tree $T$ such that
    - each node $u$ of $T$ is labelled by a subset $V_u$ of $\leq k$ vertices in $V$
    - for every edge $(v, w) \in E$ there is a node $u$ of $T$ such that both $v, w \in V_u$.
    - for every $v \in V$ the set of nodes $u$ in $T$ with $v \in V_u$ is connected in $T$
  - The tree with the sets are called the \textit{tree decomposition} of $G$.
  - The minimum $k$ and tree decomposition can be found in linear time.
  - The tree defines a natural elimination ordering for recursive algorithms on the graph.
- \textbf{Fact:} Obstacle to treewidth $k - 1$: the $k \times k$ grid graph.

Many NP-hard problems are efficiently solvable on graphs of bounded treewidth.

Treewidth also comes up in route-finding in Google Maps: Can’t run full-blown Dijkstra on the whole graph every time a user requests a route.
What to do if the problem you want to solve is NP-hard

Try to make an exponential-time solution as efficient as possible.
e.g. Try to search the space of possible hints/certificates in a more efficient way and hope that it is quick enough.

**Backtracking search**
e.g., for **SAT**, search through the $2^n$ possible truth assignments...
...but set the truth values one-by-one so we can able to figure out whole parts of the space to avoid,
e.g. Given $F = (\neg x_1 \lor x_2) \land (\neg x_2 \lor x_3) \land (x_4 \lor \neg x_3) \land (x_1 \lor x_4)$
after setting $x_1 = 1$ and $x_2 = 0$ we don’t even need to set $x_3$ or $x_4$ to know that it won’t satisfy $F$.

**Now:** More clever backtracking search for **SAT** solutions
SAT Solving

SAT is an extremely flexible problem:

• The fact that SAT is an NP-complete problem says that we can re-express a huge range of problems as SAT problems.

This means that good algorithms for SAT solving would be useful for a huge range of tasks.

Since roughly 2001, there has been a massive improvement in our ability to solve SAT on a wide range of practical instances.

• These algorithms aren’t perfect. They fail on many worst-case instances.
Satisfiability Algorithms

Local search: Solve SAT as a special case of MaxSAT
   (incomplete, may fail to find satisfying assignment)

   GSAT – random local search [Selman,Levesque,Mitchell 92]
   Walksat – Metropolis [Kautz,Selman 96]

Backtracking search (complete)
   • DPLL    [Davis,Putnam 60], [Davis,Logeman,Loveland 62]
   • CDCL: Adds clause learning and restarts
      GRASP, SATO, zchaff, MiniSAT, Glucose, etc.
CNF Satisfiability

**SAT:** satisfiability problem for CNF formulas with any clause size

Write CNFs with the $\land$ between clauses implicit:

\[ F = (x_1 \lor \overline{x_2} \lor x_4)(\overline{x_1} \lor x_3)(\overline{x_3} \lor x_2)(\overline{x_4} \lor \overline{x_3}) \]

Write assignment as literals assigned true: \(x_1, x_2, x_3, \overline{x_4}\)

**Defn:** Given partial assignment \(x_3\) where

\[ F = (x_1 \lor \overline{x_2} \lor x_4)(\overline{x_1} \lor x_3)(\overline{x_3} \lor x_2)(\overline{x_4} \lor \overline{x_3}) \]

define \(\text{simplify}(F, x_3)\) by

\[ \text{simplify}(F, x_3) = (x_1 \lor \overline{x_2} \lor x_4) \]

That is: remove satisfied clauses and remove unsatisfied literals from clauses.

**Note:** \(F\) is satisfiable iff all clauses disappear under some assignment.
Backtracking search/DPLL

\[ t \leftarrow \varepsilon \]

repeat

\textbf{select} a literal \( \ell \) (some \( x \) or \( \overline{x} \))

\[ F \leftarrow \text{simplify}(F, \ell); \ t \leftarrow \text{append}(t, \ell) \]

\textbf{while} \( F \) contains a \textbf{1}-clause \( \ell' \)

\[ F \leftarrow \text{simplify}(F, \ell'); \ t \leftarrow \text{append}(t, \ell') \]

\textbf{if} \( F \) has no clauses \textbf{return} \( t \) as satisfying assignment

\textbf{if} \( F \) has an empty clause

\textbf{backtrack} to last \textbf{free step} and flip assignment (step no longer free)

\textbf{free step}

\textbf{unit propagation}
Recursive view of DPLL (without unit propagation)

\[ \text{DPLL}(F): \]

\begin{itemize}
  \item if \( F \) is empty, report satisfiable and halt
  \item if \( F \) contains the empty clause, return
  \item else, choose a literal \( x \)
    \begin{itemize}
      \item \( \text{DPLL}(\text{simplify}(F, x)) \)
      \item \( \text{DPLL}(\text{simplify}(F, \overline{x})) \)
    \end{itemize}
\end{itemize}

with unit propagation choose \( x \) to be the literal of a 1-clause if possible
DPLL on UNSAT formula

Clauses
1. $a \lor b \lor c$
2. $a \lor \neg c$
3. $\neg b$
4. $\neg a \lor d$
5. $\neg d \lor b$

Residual Formula
Extending DPLL: Clause Learning

• When backtracking in DPLL, add new clauses corresponding to causes of failure of the search

• Added conflict clauses
  • Capture reasons of conflicts
  • Obtained via unit propagations from known ones
  • Reduce future search by producing conflicts sooner
Conflict Graph: Graph of Unit Propagations

At each conflict (derivation of them empty clause) the negations of the predecessor node labels across any cut form an implied clause.

- if clause is false then could derive $\bot$

Known Clauses

$(p \lor q \lor a)$
$(\neg a \lor \neg b \lor \neg t)$
$(t \lor \neg x_1)$
$(t \lor \neg x_2)$
$(t \lor \neg x_3)$
$(x_1 \lor x_2 \lor x_3 \lor y)$
$(x_1 \lor x_2 \lor x_3 \lor \neg y)$

Decisions

$p = false$
$q = false$
$b = true$

Learn

$(p \land q \land \neg b)$

Learn

$(x_1 \lor x_2 \lor x_3)$
Best Current SAT Solvers

Conflict-Directed Clause-Learning (CDCL) Algorithms
Minisat, Glucose, MapleSAT, CaDiCaL

They rely on many optimizations:

• No explicit computation of residual formulas, just fast calculation of the unit propagations that will happen. “watched literals”
• No explicit backtracking: New clauses always chosen to generate unit propagations higher in the tree. “asserting clauses”
• Heuristics based on learned clauses to decide what free choices to make. “VSIDS”
• Pruning of cache of learned clauses so only recently used ones are kept.
• Periodic restarting search with original formula plus learned clauses.
• etc...
Best Current SAT Solvers

Conflict-Directed Clause-Learning (CDCL) Algorithms
- Minisat, Glucose, MapleSAT, CaDiCaL

They work well on many practical formulas even with hundreds of thousands of variables or more.
- Often used in proving properties of human-produced designs.
- They are incorporated in software verification tools and a variety of automated reasoning (SMT Solvers)
- We really don’t know why they work so well.
- Definitely worth a try!

However, they provably perform very badly even on some small formulas of a few hundred or thousand variables. We have a pretty good idea why.