Local Search and SAT

Tel Aviv University
Seminar in Exponential Algorithms – Prof. Haim Kaplan
Written by Shahar Lewkowicz
Exact exponential algorithms, Fomin, Kratsch, 2010 – Chapter 8
20/05/2020
Boolean variables and assignments

• n Boolean variables $x_1, \ldots, x_n$

• A literal is a variable or the negation of a variable.

• There are $2^n$ literals: $x_1, \neg x_1, x_2, \neg x_2, \ldots, x_n, \neg x_n$

• A truth assignment (i.e. assignment) is a rule which maps each of the variables $x_1, \ldots, x_n$ to a Boolean value (0 or 1).

• An example of an assignment (when n=4):

$$x_1 = 0, \ x_2 = 1, \ x_3 = 1, \ x_4 = 0$$
Boolean disjunction formulas

• A disjunction clause (i.e. clause) of k variables: \( l_{i_1} \lor l_{i_2} \lor ... \lor l_{i_k} \) where for each \( 1 \leq j \leq k \), \( l_{i_j} \) is a literal.

• Example (when \( k=4 \), assuming \( n\geq100 \)): \( x_5 \lor \neg x_{10} \lor \neg x_{85} \lor x_{100} \)

• An assignment satisfies a clause if there exists at least one \( 1 \leq j \leq k \) such that the literal \( l_{i_j} \) is 1 according to the assignment.

• For example, the assignment \( x_1 = 0, x_2 = 0, ..., x_n = 0 \) satisfies the above clause; the assignment that gives all variables 0 except the variables \( x_{10}, x_{85} \) (which are given the value 1) does not satisfy the close.
K-SAT

• We are given $n$ variables $x_1, \ldots, x_n$

• We are given $m$ clauses (i.e. a formula)

• Each clause contains exactly $k$ literals (why this restriction?...)  

• In the decision problem, we ask whether there is at least one assignment that satisfies ALL the clauses (the formula), or not.

• Sometimes, we also wish to find an assignment that satisfies all the clauses, if there exists one.

• The latter problem is not easier than the previous. In this class – all the algorithms we will talk about solve the latter problem.
A Formula Example

- n=4, m=5, k=3 (3-SAT)

\[
\varphi = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_3 \lor \neg x_4) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (x_2 \lor x_3 \lor x_4) \land (x_1 \lor x_2 \lor \neg x_4)
\]

- Satisfying assignment: \( x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0 \)

- Non satisfying assignment: \( x_1 = 1, x_2 = 1, x_3 = 1, x_4 = 1 \)
Assignments and n-Bit Strings

• We can declare a function that maps each assignment
  \[ x_1 = b_1, \ x_2 = b_2, \ \ldots, x_n = b_n \] to the n-bit string \[ b_1b_2\ldots b_n \]
• This function is a bijection, so we can also speak about it’s inverse function.
• \[ | \{ a \mid a \text{ is an assignment} \} | = | \{0,1\}^n | = | \{ t \mid 0 \leq t \leq 2^n - 1 \} | = 2^n \]
Hamming Distance - Definition

• Given two n bit strings $b_1b_2...b_n$, $c_1c_2...c_n$ we define the hamming distance between them:

$$d(b_1b_2...b_n , c_1c_2...c_n) = \sum_{i=1}^{n} |c_i - b_i| = \sum_{i=1}^{n} b_i \oplus c_i$$

• Given two assignments on n variables b and c we define the hamming distance between them to be the hamming distance between their two representatives n-bit strings.
Hamming Distance – Example and Facts

• For example: the hamming distance between these two assignments:

\[
x_1 = 0, \ x_2 = 0, \ x_3 = 1, \ x_4 = 1, \ x_5 = 0 \quad \text{and} \quad x_1 = 1, \ x_2 = 0, \ x_3 = 1, \ x_4 = 0, \ x_5 = 0
\]

is the hamming distance between these two bit strings:

00110 \quad \text{and} \quad 10100 \quad \text{which is 2.}

• It can be seen that:

\[
d(a,b) = d(b,a)
\]

\[
0 \leq d(a,b) \leq n
\]

\[
d(a,b) = 0 \iff a = b
\]
Balls - Definition

• Given an assignment $a$ and an integer $r$, we define:

$$B(a, r) = \{ s \mid s \text{ is an assignment } \land d(s, a) \leq r \}$$

• It can be seen that

$$|B(a, r)| = \sum_{i=0}^{r} \binom{n}{i}$$

• Note that the size above is not dependent on the assignment $a$ (i.e. The size of the ball is the same for all assignments).
Balls – Example in case n=4: B(0000, 1)
Algorithms solving K-SAT

- It takes $O(mk)$ time to check whether a given assignment satisfies a given formula (why?)
- Therefore - instead of analyzing the time of an algorithm – we will analyze the number of the assignments that the algorithm checks.
The Naïve Algorithm - Definition

• Input: a formula \( f \).
• Output: Answer whether \( f \) is satisfiable or not; if it is – also output an assignment that satisfies it.
• Algorithm:
  • 1. For each possible assignment \( a \):
    • 1.1. Check if \( a \) satisfies \( f \). If it is:
      • 1.1.1. output \( a \) and conclude that \( f \) is satisfiable.
    • 2. Conclude that \( f \) is not satisfiable.
The Naïve Algorithm - Analyze

• If the formula is not satisfiable – the time is always $2^n$.

• If the formula is satisfiable by one assignment:
  • The time in the worst case is $2^n$
  • The time in the average case (assuming...) is $2^n / 2 = 2^{n-1}$.

• This is huge! We want to reduce this time!
Geometric Distribution - Definition

• Let’s assume we flip a coin which falls on 1 with probability $0 \leq p \leq 1$ (i.e. success) and falls on 0 with probability $1-p$ (i.e. failure).
• We repeat flipping the coin until we succeed (i.e. until coin falls on 1).
• Let $N$ be the number of times we had to flip the coin (including the last flipping – which succeeded).
• We will say that $N$ is a geometric variable with the parameter $p$.
• It can be proven that $E[N] = \frac{1}{p}$
Markov’s Inequality

Let \( X \) be a non-negative random variable and let \( c > 0 \). Then:

\[
\Pr[X > c] \leq \frac{E[X]}{c}
\]

\( c = 2E[X], \Pr \leq 1/2 \)
Random_Walk Algorithm - Definition

1. Choose a truth assignment \( a \) - uniformly at random.
2. \( \text{count} \leftarrow 0 \)
3. while \( \text{count} < 3n \):
   3.1. Choose an arbitrary clause which is not satisfied by \( a \).
   3.2. Choose one variable \( v \) of this clause - uniformly at random.
   3.3. Flip \( v \)'s value (in the assignment \( a \)).
   3.4. if \( a \) is a satisfying assignment:
      3.4.1. Output \( a \) and conclude that \( f \) is satisfiable.
   3.5. \( \text{count} \leftarrow \text{count} + 1 \)
4. Conclude that \( f \) is not satisfiable.
Random_Walk Algorithm - Analyzing

• Time **in the worst case** is 3n=O(n).

• If the formula is not satisfiable – the algorithm will always be correct.

• If the formula is satisfiable – what is the probability that the algorithm will be correct? We would like to find a lower bound...

• Assuming the formula is satisfiable, let’s find a lower bound on q – the probability that the algorithm will be correct (in this case).

• There is at least one satisfying assignment a* of the formula. We need to find a lower bound on the probability that the algorithm will output a* (why?)
Random_Walk Algorithm – Analyzing (2)

• At each step, the hamming distance from \(a^*\) to \(a\) decreases by 1 (good move) with probability of at least \(1/k\) and increased by 1 (bad move) with probability of at least \((k-1)/k\). (why?)
Random_Walk Algorithm – Analyzing (3)

• In the following 2 slides, let’s assume that we start from an assignment $a$ which is at distance $j$ from $a^*$:

• We define $q_{j,x}$ the probability that the algorithm finds $a^*$ in $j+2x$ steps: $j+x$ steps are “good” moves and $x$ steps are “bad” moves.

• $q_{j,x}$ is a conditional probability.

$$q_{j,x} \geq \binom{j+2x}{x} \cdot \left( \frac{k-1}{k} \right)^x \cdot \left( \frac{1}{k} \right)^{j+x}$$
Random_Walk Algorithm – Analyzing (4)

• Assuming we start from an assignment $a$ which is at distance $j$ from $a^*$:

• Define $q_j$ the probability that the algorithm finds $a^*$ when starting from an assignment $a$ which is at distance $j$ from $a^*$.

• $q_j$ is a conditional probability.

$$q_j \geq \max_{0 \leq x \leq (3n-j)/2} q_{j,x} \geq \frac{1}{\sqrt{8j}} \left( \frac{1}{k-1} \right)^j$$

• In the right inequation, we choose $x = \frac{j}{k-2}$ and we cannot do better...
Random_Walk Algorithm – Analyzing (5)

• Let $p_j$ be the probability that we choose a to be at distance $j$ from $a^*$

$$p_j = \binom{n}{j} \left( \frac{1}{2} \right)^n$$

• Let $q$ be the probability that the algorithm successfully find $a^*$.

$$q \geq \sum_{j=0}^{n} p_j q_j \geq \cdots \geq \frac{1}{\sqrt{8n}} \cdot \left( \frac{k}{2 \cdot (k-1)} \right)^n$$
Persistent_Walk Algorithm

- **Assumption**: the formula is satisfiable.
- **The algorithm**: run Random_Walk until it returns a satisfying assignment.
- The algorithm is always correct. The probability for infinite loop: 0.
- Let N be the number of times Random_Walk was called.

\[ E[N] \leq \frac{1}{q} = \sqrt{8n \cdot \left(\frac{2\cdot(k-1)}{k}\right)^n} \]

- The expected time of Persistent_Walk: \( O(n^{1.5} \cdot \left(\frac{2\cdot(k-1)}{k}\right)^n) \)
Persistent_Walk vs the Naïve Algorithm

• Advantages of Persistent_Walk:
  • Estimated time is much lower than Naïve Algorithm.
  • We do not depend on the mercy of an evil opponent (why?)

• Advantages of Naïve Algorithm:
  • No assumption that the formula is satisfiable.
  • The actual running time is NEVER bigger than $2^n$. 
Half_Monte_Carlo Algorithm - Definition

1. \( \text{count} \leftarrow 0 \)

2. while \( \text{count} < 2 \cdot \left\lfloor \frac{1}{q} \right\rfloor \):
   
   2.1. Run Random_Walk()
   
   2.2. if the above call concluded that \( f \) is satisfiable:
       
       2.2.1. Output the printed assignment and conclude that \( f \) is satisfiable.

   2.3. \( \text{count} \leftarrow \text{count} + 1 \)

3. Conclude that \( f \) is not satisfiable.
Half_Monte_Carlo Algorithm - Analyzing

• The worst case time: \( O\left(\frac{n}{q}\right) = O(n^{1.5} \left(\frac{2\cdot(k-1)}{k}\right)^n) \)

• If the formula is not satisfiable – Half_Monte_Carlo will always be correct.

• If the formula is satisfiable - Half_Monte_Carlo will conclude that it is not with probability of at most 0.5. We will prove it now.

• Let \( N \) be the number of times Random_Walk should be called in order to find a satisfying assignment.

• As we have seen, \( E[N] \leq \frac{1}{q} \)
Half_Monte_Carlo Algorithm – Analyzing (2)

- \( E[N] \leq \frac{1}{q} \) but error occurs when \( N > 2 \cdot \left\lfloor \frac{1}{q} \right\rfloor \) so from Markov (c=2)

we get \( \Pr[error] \leq \frac{1}{2} \)

- But what if we want to lower the error probability?
- If we run this scheme b times – we will get an error probability of \( \frac{1}{2^b} \)

- Credit to Schoning
What we have seen so far?

• We have seen an example of local search: using random walk.

• We have seen the Persistent_Walk algorithm:
  • The assumption that the formula is satisfiable is required.
  • The algorithm is always correct.
  • The estimated running time is \( O(n^{1.5} \cdot \left( \frac{2 \cdot (k - 1)}{k} \right)^n) \).

• We have seen the Half_Monte_Carlo algorithm:
  • The algorithm has a one-sided error of \( \frac{1}{2} \).
  • The running time in the worst case is the same as above: \( O(n^{1.5} \cdot \left( \frac{2 \cdot (k - 1)}{k} \right)^n) \).
Ball_Search - Definition

- Input: a formula $f$, a truth assignment $a$, a non-negative integer $d$.
- Output: Answer whether there exists a satisfiable assignment $a^*$ such that $a^* \in B(a,d)$; if there is – also output such an assignment.
- Algorithm:
Ball_Search Algorithm - Definition (2)

• 1. Check if $a$ satisfies $f$. If it is:
  • 1.1. output $a$ and return “yes”.
• 2. if $d=0$:
  • 2.1. return “no”.
• 3. Choose an arbitrary clause which is not satisfied by $a$.
• 4. for each literal in that clause:
  • 4.1. let $b$ be the truth assignment of $a$ after flipping that literal.
  • 4.2. run Ball_Search($f$, $b$, $d-1$).
  • 4.3. if the above call returned “yes”:
    • 4.3.1. output the printed assignment and return “yes”.
• 5. return “no”.
Ball_Search Algorithm - Analyzing

• Correctness:
  • It is easy to see that the algorithm is correct for $d=0$.
  • If $d>0$, $a$ is not a satisfying assignment and there is $a^* \in B(a,d) \Rightarrow$ then $a$ and $a^*$ differ in at least one variable chosen in line 3 (why?).
  • If we understand that – then it is easy to prove the correctness (using induction on $d$).

• Running time: $k^d n^{O(1)}$
Random_Ball

- Algorithm: Ball_Search(f, a, \( \alpha n \)) where a is chosen uniformly at random and \( \alpha > 0 \).
- If the formula is not satisfiable then the algorithm is correct.
- If the formula is satisfiable then there is a satisfying assignment \( a^* \).
- Therefore,
  \[
  \Pr[\text{success}] \geq \Pr[a^* \in B(a, \alpha n)] = \frac{|B(a, \alpha n)|}{|\{0,1\}^n|} = \sum_{i=1}^{\alpha n} \binom{n}{i} \frac{1}{2^n}
  \]
- We will choose \( \alpha = \frac{1}{k+1} \)
Persistent_Ball

- Same idea as in Persistent_Walk but instead of running Random_Walk – we run Random_Ball.
- Analyze is similar: The expected number of calls to Persistent_Ball
  \[
  \frac{2^n}{\sum_{i=1}^{\alpha n} \binom{n}{i}}
  \]
  so the expected running time is
  \[
  O^*\left(\frac{2^n}{\sum_{i=1}^{\alpha n} \binom{n}{i}} \cdot k^{\alpha n}\right) = O^*\left((\frac{2k}{k+1})^n\right)
  \]
- Note: we chose \(\alpha = \frac{1}{k+1}\).
- That is not better than Persistent_Walk...
Monta Carlo Algorithm for Random_Ball

• We can obtain a Monta Carlo algorithm with the same running time in a similar way like we did before...
• That algorithm will not be good as the algorithm we saw before...
Two_Front Algorithm

• The algorithm: run $\text{Ball\_Search}(f, 0^n, n/2)$ and $\text{Ball\_Search}(f, 1^n, n/2)$. Act according to the returned (and printed) value of these 2 calls...
Two_Front Algorithm - Analyze

• Correctness: The algorithm is correct because for each possible assignment $a$: $d(a, 0^n) \leq n/2$ or $d(a, 1^n) \leq n/2$.

• Running time: $k^{n/2}n^{O(1)}$

• If $k=3$ (3-SAT) then the running time is $3^{n/2}n^{O(1)} = \sqrt{3^n} \cdot n^{O(1)}$. That is better than the naïve algorithm!

• What if $k>3$? That is worse than the naïve algorithm. Perhaps we should try to improve it by using balls of smaller sizes...
What we would like to do?
Reminder: Dominating Set

• Dominating Set $D$ in graph $G=(V,E)$ is a subset of $V$ such that for each $u \in V$: either $u \in D$ or $u$ has a neighbor $w$ such that $w \in D$.

• In other words, for each $u \in V$ there exists $w \in D$ such that $u \in B(w,1)$.

• 3 Examples:
Another Example of Dominating Set

\[ V = \{0, 1\}^n \]
\[ E = \{(a, b) \in V \times V \mid d(a, b) = 1\} \]
\[ G = (V, E) \]
\[ D = \{0, 1\}^{n-1} \cup \{0\} \]
Cover Code - Definition

• A cover code $C$ of radius $r$ (in $\{0,1\}^n$) is a subset of $\{0,1\}^n$ such that for each truth assignment $a \in \{0,1\}^n$ there is $b \in C$ such that $d(a,b) \leq r$.

• For example, $C = \{0,1\}^{n-3} \cdot \{000\}$ is a code of radius 3 (why?)

• Dominating Set is a code cover of radius 1.

• Our target – to make a deterministic version of Persistent_Ball without increasing the expected time using a code cover! How?
Reminder: Minimum Set Cover

- Universe \( U = \{1, 2, \ldots, u\} \).
- A collection of sets \( S = \{S_1, S_2, \ldots, S_r\} \) such that \( \forall 1 \leq i \leq t : S_i \subseteq U \).
- We need to find a subset of \( S \) \( A = \{A_1, A_2, \ldots, A_q\} \subseteq S \) with minimum number of sets (i.e. minimize \( q \)) such that for each \( j \in U \) there exists a set \( M \in A \) such that \( j \in M \).

- Greedy Algorithm: Always pick the set which covers the maximum number of uncovered elements, until all elements are covered.
- The Greedy Algorithm is \( \log t \)-approximation.
Minimum Set Cover - Example
Reduction from Cover Code to Set Cover

- The universe $U$ will be the possible assignments.
- The number of sets (i.e. $t$, the size of $S$) will also be $2^n$ - for each possible assignment $a$ – there will be a suitable set $B(a, an)$.
Back to Code Cover

• If $C$ is a code cover of radius $\alpha n$, $0 < \alpha \leq 1/2$, then $|C| \geq \frac{2^n}{|B(0^n, \alpha n)|}$

• There exists a code $C$ of radius $\alpha n$ such that $|C| \leq \left[ n \cdot \frac{2^n}{|B(0^n, \alpha n)|} \right]$

• Therefore, the Greedy Algorithm for Set Cover will output a cover code of radius $\alpha n$ such that the code’s size will be $O\left( n^2 \cdot \frac{2^n}{|B(0^n, \alpha n)|} \right)$
What is left to do?

1. Prove the claim: There exists a code $C$ of radius $\alpha n$ such that

$$|C| \leq \left\lfloor n \cdot \frac{2^n}{|B(0^n, \alpha n)|} \right\rfloor$$

2. Speak about the running time of our algorithm to find the code cover. If it is too high – we will want to find ways to reduce it...
Probabilistic Proof

• We will choose uniformly at random \( \left\lfloor n \cdot \frac{2^n}{|\mathcal{B}(0^n, \alpha n)|} \right\rfloor \) assignments (with possible repetitions).

• We will show that the probability that \( C \) is a cover code of radius \( \alpha n \) is bigger than zero – which means such code exists (if there wasn’t such a code – the probability would have been zero).

• Let \( a \) be an assignment. For each \( c \in C \):

\[
\Pr[d(a, c) > \alpha n] = 1 - \Pr[d(a, c) \leq \alpha n] = 1 - \frac{|\mathcal{B}(c, \alpha n)|}{2^n} = 1 - \frac{|\mathcal{B}(0^n, \alpha n)|}{2^n}
\]

• The probability that \( a \) is not covered by \( C \) is \( (1 - \frac{|\mathcal{B}(0^n, \alpha n)|}{2^n})^{|C|} \leq e^{-n} \).
Probability Proof (2)

• The probability that C is **not** a cover code with radius \( \alpha n \) is
  \[
  \Pr[\text{there is a which is not covered}] \leq \sum_{a \in \{0,1\}^n} \Pr[a \text{ is not covered}] \leq \\
  \leq \sum_{a \in \{0,1\}^n} e^{-n} = 2^n \cdot e^{-n} = \left(\frac{2}{e}\right)^n
  \]

• The probability that C is a cover code with radius \( \alpha n \) is at least \( 1 - \left(\frac{2}{e}\right)^n > 0 \)
• Therefore, there exists such a code.
Running Time of the Algorithm for Code Cover

- The number of selections is at most $2^n$ and for every selection a re-computation of covered assignments takes time $O^*(4^n)$.

- Therefore: For each $0 < \alpha \leq 1/2$ a covering code $C$ of radius $\alpha n$ of size of at most $O\left( n^2 \cdot \frac{2^n}{|B(0^n, \alpha n)|} \right)$ can be computed in time $O^*(8^n)$.

- Problem: But this is too much time! Can we improve it?
Improve the Running Time

• By partition the set of n variables into 6 groups of size n/6 and take care of each group separately.

• What will we do if n is not divided by 6?

• We will solve the problem for each of this 6 groups of variables in order to get 6 cover codes (each with radius $\alpha n/6$).

• Then we will try all the possible concatenations of codes in order to get the code that we want.

• The total running time in the worst case will be $O^*\left(\left(\frac{2k}{k+1}\right)^n\right)$.

• This is also the expected running time of Persistent_Ball!
Conclusion

• We have seen the basic idea of local search.
• We have seen 2 types of local search: random walk (random path) and ball search.
• Walking is better than balls in regarding the expected time.
• However, by ball search we can obtain deterministic algorithms which have estimated time that may not be good as the walking, but is still much better than the naïve algorithm.