GEMS OF TCS

HEURISTIC ALGORITHMS

Sasha Golovnev
February 25, 2021
Announcements

1. Deadline HW2 today

2. Next Tuesday:
   Nitin Vaidya

3. Next Thursday:
   Last lecture on Algs

4. 3rd HW will be posted
   next Thursday

5. After this:
   Complexity
   Crypto
   Learning

6. Post-Quantum Crypto talk:
   Noah SD (Cornell), Mar 5, 1pm
Heuristic Algorithms

- When exact algorithms are too slow, and approximate algorithm are not accurate enough
Heuristic Algorithms

- When exact algorithms are too slow, and approximate algorithm are not accurate enough
- We can use heuristic algorithms
Heuristic Algorithms

• When exact algorithms are too slow, and approximate algorithm are not accurate enough
• We can use heuristic algorithms
• Heuristic algorithms use practical methods that are not guaranteed/proved to be optimal or efficient
HEURISTIC ALGORITHMS

- When exact algorithms are too slow, and approximate algorithm are not accurate enough
- We can use heuristic algorithms
- Heuristic algorithms use practical methods that are not guaranteed/proved to be optimal or efficient
- Some heuristic algorithms are fast but not guaranteed to find optimal solutions
Heuristic Algorithms

- When exact algorithms are too slow, and approximate algorithm are not accurate enough
- We can use heuristic algorithms
- Heuristic algorithms use practical methods that are not guaranteed/proved to be optimal or efficient
- Some heuristic algorithms are fast but not guaranteed to find optimal solutions
- Some heuristic algorithms find optimal solutions but not guaranteed to be fast
Traveling Salesman
TRAVELING SALESMAN PROBLEM

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once.
**Traveling Salesman Problem**

Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once

![Graph with nodes and edges labeled with weights. The highlighted cycle has a length of 9.]
NEAREST NEIGHBORS

• Going to the nearest unvisited node at every iteration?
Nearest Neighbors

- Going to the nearest unvisited node at every iteration?
- Efficient, works reasonably well in practice $O(n^2)$
Nearest Neighbors

- Going to the nearest unvisited node at every iteration?
- Efficient, works reasonably well in practice
- For general graphs, may produce a cycle that is much worse than an optimal one
Nearest Neighbors

- Going to the nearest unvisited node at every iteration?
- Efficient, works reasonably well in practice
- For general graphs, may produce a cycle that is much worse than an optimal one
- For Euclidean instances, the resulting cycle may be about $\log n$ times worse than an optimal one
Nearest Neighbors: Bad Case

• How to fool the nearest neighbors heuristic?
Nearest Neighbors: Bad Case

• How to fool the nearest neighbors heuristic?
• Assume that the weights of almost all the edges in the graph are equal to 2
Nearest Neighbors: Bad Case

• How to fool the nearest neighbors heuristic?
• Assume that the weights of almost all the edges in the graph are equal to 2
• And we start to construct a cycle:
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:
Nearest Neighbors: Bad Case

- How to fool the nearest neighbors heuristic?
- Assume that the weights of almost all the edges in the graph are equal to 2
- And we start to construct a cycle:
Nearest Neighbors: Bad Case

• How to fool the nearest neighbors heuristic?
• Assume that the weights of almost all the edges in the graph are equal to 2
• And we start to construct a cycle:
Suboptimal Solution for Euclidean TSP

\[ \sqrt{9+9} = \sqrt{18} \]
SUBOPTIMAL SOLUTION FOR EUCLIDEAN TSP

OPT ≈ 26.42
Suboptimal Solution for Euclidean TSP

OPT ≈ 26.42
Suboptimal Solution for Euclidean TSP

Opt $\approx 26.42$
Suboptimal Solution for Euclidean TSP

OPT ≈ 26.42
SUBOPTIMAL SOLUTION FOR EUCLIDEAN TSP

OPT $\approx 26.42$
Suboptimal Solution for Euclidean TSP

$\text{OPT} \approx 26.42$

$\text{NN} \approx 28.33$
LOCAL SEARCH

Another Heuristic

- $s \leftarrow$ some initial solution — any cycle that visits each vertex exactly once: $1 \rightarrow 2 \rightarrow 3 \rightarrow \ldots \rightarrow n$
LOCAL SEARCH

- $s \leftarrow$ some initial solution
- while it is possible to change 2 edges in $s$ to get a better cycle $s'$:
Local Search

- $s \leftarrow$ some initial solution
- while it is possible to change 2 edges in $s$ to get a better cycle $s'$:
  - $s \leftarrow s'$
LOCAL SEARCH

• $s \leftarrow$ some initial solution
• while it is possible to change 2 edges in $s$ to get a better cycle $s'$:
  • $s \leftarrow s'$
• return $s$
EXAMPLE

Changing two edges in a suboptimal solution:
EXAMPLE

Changing two edges in a suboptimal solution:
Example

Changing two edges in a suboptimal solution:
Changing two edges in a suboptimal solution:
EXAMPLE

A suboptimal solution that cannot be improved by changing two edges:
A suboptimal solution that cannot be improved by changing two edges:

Need to allow changing three edges to improve this solution
Local Search with parameter $d$:

- $s \leftarrow$ some initial solution
- while it is possible to change $d$ edges in $s$ to get a better cycle $s'$:
  - $s \leftarrow s'$
- return $s$
Properties

• Computes a local optimum instead of a global optimum
• Computes a local optimum instead of a global optimum
• The larger $d$, the better the resulting solution and the higher is the running time

$d=2$  \hspace{1cm} \text{pairs of edges} \hspace{1cm} O(n^2)$  
$d=10$  \hspace{1cm} \text{10-tuples of edges} \hspace{1cm} O(n^{10})$
Performance

• Trade-off between quality and running time of a single iteration
Performance

- Trade-off between quality and running time of a single iteration
- Still, the number of iterations may be exponential and the quality of the found cycle may be poor
Performance

- Trade-off between quality and running time of a single iteration
- Still, the number of iterations may be exponential and the quality of the found cycle may be poor
- But works well in practice
Satisfiability
\[(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3)\]
SAT

1. $x_1 \lor x_2 = x_3$
   
   $(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3)$

2. UNSAT
   
   $(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3)$

   *We've seen exponential algorithms*
   
   *Well heuristics algorithms*
BACKTRACKING

\[ 2^n \text{ - trivial alg.} \]

\[ x_1, x_2, \ldots, x_n \]

- Construct a solution piece by piece
Backtracking

- Construct a solution piece by piece
- Backtrack if the current partial solution cannot be extended to a valid solution
Backtracking

\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]
EXAMPLE

\[(x \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x \lor x_2 \lor \neg x_3)(\neg x_2)(x_2 \lor \neg x_4)\]

\[x_1 = 0\]
EXAMPLE

\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]

\[x_1 = 0\]

\[(x_2 \lor x_3 \lor x_4)(x_3 \lor \neg x_3)(\neg x_2)(\neg x_2 \lor \neg x_4)\]

\[x_2 = 0\]

\[(x_3 \lor x_4)(\neg x_3)(\neg x_4)\]
\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]

- \(x_1 = 0\)
- \(x_2 = 0\)
- \(x_3 = 0\)
- \(x_4\)
\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]

- If \(x_1 = 0\):
  \[(x_2 \lor x_3 \lor x_4)(x_2 \lor \neg x_3)(\neg x_2)(\neg x_4)\]
  - If \(x_2 = 0\):
    \[(x_3 \lor x_4)(\neg x_3)(\neg x_4)\]
    - If \(x_3 = 0\):
      \[(\neg x_4)(\neg x_4)\]
  - If \(x_2 = 0\):
    \[(x_2 \lor \neg x_3)(\neg x_2)(\neg x_4)\]
- If \(x_1 = 0\):
  \[(x_2 \lor x_3 \lor x_4)(x_2 \lor \neg x_3)(\neg x_2)(\neg x_4)\]

\[\text{UNSAT}\]
EXAMPLE

\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]

\[x_1 = 0\]

\[(x_2 \lor x_3 \lor x_4)(x_2 \lor \neg x_3)(\neg x_2)(x_2 \lor \neg x_4)\]

\[x_2 = 0\]

\[(x_3 \lor x_4)(\neg x_3)(\neg x_4)\]

\[x_3 = 0\]

\[(x_4)(\neg x_4)\]

\[x_4 = 0\]

\[x_4 = 1\]

() ()
\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]

**Example**

\[(x_2 \lor x_3 \lor x_4)(x_2 \lor \neg x_3)(\neg x_2)(x_2 \lor \neg x_4)\]

- **x_1 = 0**

\[(x_3 \lor x_4)(\neg x_3)(\neg x_4)\]

- **x_2 = 0**

\[(x_4)(\neg x_4)\]

- **x_3 = 0**

\[(())(\neg x_4)\]

- **x_3 = 1**

\[(x_4 = 0)\]

\[(x_4 = 1)\]

- **x_4 = 0**

- **x_4 = 1**
EXAMPLE

\[(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)\]

- If \(x_1 = 0\):
  - \((x_2 \lor x_3 \lor x_4)(x_2 \lor \neg x_3)(\neg x_2)(x_2 \lor \neg x_4)\)
  - If \(x_2 = 0\):
    - \((x_3 \lor x_4)(\neg x_3)(\neg x_4)\)
      - If \(x_3 = 0\):
        - \((x_4)(\neg x_4)\)
          - If \(x_4 = 0\):
            - \((\)\)
          - If \(x_4 = 1\):
            - \((\)\)
    - If \(x_2 = 1\):
      - \((\)\)
- If \(x_2 = 1\):
  - \((x_3 \lor x_4)(\neg x_3)(\neg x_4)\)
    - If \(x_3 = 0\):
      - \((x_4)(\neg x_4)\)
        - If \(x_4 = 0\):
          - \((\)\)
        - If \(x_4 = 1\):
          - \((\)\)
    - If \(x_3 = 1\):
      - \((\)\)

\[2^n\]
EXAMPLE

\[
(x_1 \lor x_2 \lor x_3 \lor x_4)(\neg x_1)(x_1 \lor x_2 \lor \neg x_3)(x_1 \lor \neg x_2)(x_2 \lor \neg x_4)
\]

\[
\text{if } x_1 = 0
\]
\[
(x_2 \lor x_3 \lor x_4)(x_2 \lor \neg x_3)(\neg x_2)(x_2 \lor \neg x_4)
\]

\[
\text{if } x_1 = 1
\]
\[
() (x_2 \lor \neg x_4)
\]

\[
\text{if } x_2 = 0
\]
\[
(x_3 \lor x_4)(\neg x_3)(\neg x_4)
\]

\[
\text{if } x_2 = 1
\]
\[
() (\neg x_4)
\]

\[
\text{if } x_3 = 0
\]
\[
(x_4)(\neg x_4)
\]

\[
\text{if } x_3 = 1
\]
\[
() (\neg x_4)
\]

\[
\text{if } x_4 = 0
\]
\[
() (\neg x_4)
\]

\[
\text{if } x_4 = 1
\]
\[
() (\neg x_4)
\]
Backtracking Algorithm

- SolveSAT($F$):
  - if $F$ has no clauses:
    return “sat”
  - if $F$ contains an empty clause:
    return “unsat”
Backtracking Algorithm

- **SolveSAT**(*F*):
  - if *F* has no clauses:
    return “sat”
  - if *F* contains an empty clause:
    return “unsat”
  - *x* ← unassigned variable of *F*
BACKTRACKING ALGORITHM

• SolveSAT($F$):
  • if $F$ has no clauses:
    return “sat”
  • if $F$ contains an empty clause:
    return “unsat”
  • $x \leftarrow$ unassigned variable of $F$
  • if $\text{SolveSAT}(F[x \leftarrow 0]) = “sat”$:
    return “sat”
BACKTRACKING ALGORITHM

- SolveSAT(F):
  - if F has no clauses:
    return “sat”
  - if F contains an empty clause:
    return “unsat”
  - x ← unassigned variable of F
  - if SolveSAT(F[x ← 0]) = “sat”:
    return “sat”
  - if SolveSAT(F[x ← 1]) = “sat”:
    return “sat”
  - if no solutions with x=0 on
    x=1, then
  there are no solutions ⇒ flag is UNSAT
**Backtracking Algorithm**

- SolveSAT($F$):
  - if $F$ has no clauses:
    return “sat”
  - if $F$ contains an empty clause:
    return “unsat”
  - $x \leftarrow$ unassigned variable of $F$
  - if SolveSAT($F[x \leftarrow 0]$) = “sat”:
    return “sat”
  - if SolveSAT($F[x \leftarrow 1]$) = “sat”:
    return “sat”
  - return “unsat”
\[
\begin{align*}
\left( x_1 \right) \left( \neg x_1 \right) \left( x_2 \lor x_3 \right) \left( x_7 \lor x_{10} \right)
\end{align*}
\]
• Thus, instead of considering all $2^n$ branches of the recursion tree, we track carefully each branch
Backtracking

• Thus, instead of considering all $2^n$ branches of the recursion tree, we track carefully each branch.
• When we realize that a branch is dead (cannot be extended to a solution), we immediately cut it.
SAT Solvers

- Backtracking is used in many state-of-the-art SAT-solvers
SAT SOLVERS

• Backtracking is used in many state-of-the-art SAT-solvers
• SAT-solvers use tricky heuristics to choose a variable to branch on, simplify a formula before branching, and use efficient data structures

Example: choose a var that appears more often
Example: \( x = 0 \) or \( x = 1 \) first?
Simplify: \((x, \sqrt{x_2})(x_3)(x_4 \lor \sqrt{x_3})\)
\( x_3 = 1 \)
SAT Solvers

- Backtracking is used in many state-of-the-art SAT-solvers
- SAT-solvers use tricky heuristics to choose a variable to branch on, simplify a formula before branching, and use efficient data structures
- Another commonly used technique is local search

\[(x_1 \lor x_2 \lor x_3) - \text{this clause is currently unsat}
\]

\[
x_1 = 0 \quad x_2 = 1 \quad x_3 = 0
\]

Change value of one of these vars SAT
Applications
THE ART OF
COMPUTER PROGRAMMING

VOLUME 4    PRE-FASCICLE 6A

A DRAFT OF
SECTION 7.2.2.2:
SATISFIABILITY

DONALD E. KNUTH    Stanford University
Wow! — Section 7.2.2.2 has turned out to be the longest section, by far, in *The Art of Computer Programming*. The SAT problem is evidently a “killer app,” because it is key to the solution of so many problems. Consequently I can only hope that my lengthy treatment does not also kill off my faithful readers!

Donald Knuth
Annual SAT Conference (since 1996): http://satisfiability.org
Conference, Competition, Journal

- Annual SAT Conference (since 1996): http://satisfiability.org
- Annual SAT Solving competitions (since 2002): http://www.satcompetition.org/
Conference, Competition, Journal

- Annual SAT Conference (since 1996): http://satisfiability.org
- Annual SAT Solving competitions (since 2002): http://www.satcompetition.org/
- Journal on Satisfiability, Boolean Modeling and Computation: http://jsatjournal.org/
Two-hundred-terabyte maths proof is largest ever
A computer cracks the Boolean Pythagorean triples problem — but is it really maths?

Evelyn Lamb

28 May 2016
**Geometry**

**Computer Search Settles 90-Year-Old Math Problem**

By translating Keller’s conjecture into a computer-friendly search for a type of graph, researchers have finally resolved a problem about covering spaces with tiles.

$d = 7$-dim space
from pycosat import solve

clauses = [[-1, -2, -3], [1, -2], [2, -3], [3, -1], [1, 2, 3]]

print(solve(clauses))

print(solve(clauses[1:]))
```python
from pycosat import solve

clauses = [ [-1, -2, -3], [1, -2], [2, -3], [3, -1], [1, 2, 3] ]

print(solve(clauses))
print(solve(clauses[1:]))

UNSAT
[1, 2, 3] \rightarrow x_1=1 \ x_2=1 \ x_3=1

[-1, 2, -3] \rightarrow x_1=0 \ x_2=1 \ x_3=0
```
Is it possible to place $n$ queens on an $n \times n$ board such that no two of them attack each other?
Examples
Examples

Classical solution:
Brute force: even \( n=8 \)
Backtracking: place 1st queen

Encode/reduce to SAT
use SAT-solvers

(64) ways too large
\( n \approx 20 \)
ENCODING AS SAT

- $n^2$ 0/1-variables: for $0 \leq i, j < n$, $x_{ij} = 1$ iff queen is placed into cell $(i, j)$
ENCODING AS SAT

- $n^2$ 0/1-variables: for $0 \leq i, j < n$, $x_{ij} = 1$ iff a queen is placed into cell $(i, j)$
- For $0 \leq i < n$, $i$th row contains $\geq 1$ queen:
  \[
  (x_{i1} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1).
  \]
**Encoding as SAT**

- $n^2$ 0/1-variables: for $0 \leq i, j < n$, $x_{ij} = 1$ iff queen is placed into cell $(i, j)$
- For $0 \leq i < n$, $i$th row contains $\geq 1$ queen:
  $$\overline{(x_{i1} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1)}.$$
- For $0 \leq i < n$, $i$th row contains $\leq 1$ queen:
  $$\forall 0 \leq j_1 \neq j_2 < n: \ (x_{ij_1} = 0 \text{ or } x_{ij_2} = 0).$$

\[(x_{0,0} = 0 \text{ or } x_{0,10} = 0) \text{ It's not true}\]

\[\neg (x_{0,0} \text{ or } x_{0,10})\]
**ENCODING as SAT**

- \(n^2\) 0/1-variables: for \(0 \leq i, j < n\), \(x_{ij} = 1\) iff queen is placed into cell \((i, j)\)
- For \(0 \leq i < n\), \(i\)th row contains \(\geq 1\) queen:
  \[
  (x_{i1} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1).
  \]
- For \(0 \leq i < n\), \(i\)th row contains \(\leq 1\) queen:
  \[
  \forall 0 \leq j_1 \neq j_2 < n: \ (x_{ij_1} = 0 \text{ or } x_{ij_2} = 0).
  \]
- For \(0 \leq j < n\), \(j\)th column contains \(\leq 1\) queen:
  \[
  \forall 0 \leq i_1 \neq i_2 < n: \ (x_{i_1j} = 0 \text{ or } x_{i_2j} = 0).
  \]
ENCODING AS SAT

- $n^2$ 0/1-variables: for $0 \leq i, j < n$, $x_{ij} = 1$ iff queen is placed into cell $(i, j)$
- For $0 \leq i < n$, $i$th row contains $\geq 1$ queen:
  $$(x_{i1} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1).$$
- For $0 \leq i < n$, $i$th row contains $\leq 1$ queen:
  $\forall 0 \leq j_1 \neq j_2 < n: \ (x_{ij_1} = 0 \text{ or } x_{ij_2} = 0).$
- For $0 \leq j < n$, $j$th column contains $\leq 1$ queen:
  $\forall 0 \leq i_1 \neq i_2 < n: \ (x_{i_1j} = 0 \text{ or } x_{i_2j} = 0).$
- For each pair $(i_1, j_1), (i_2, j_2)$ on diagonal:
  $$(x_{i_1j_1} = 0 \text{ or } x_{i_2j_2} = 0).$$
from itertools import combinations, product
from pycosat import solve

n = 10
clauses = []

# converts a pair of integers into a unique integer
def varnum(i, j):
    assert i in range(n) and j in range(n)
    return i * n + j + 1

# each row contains at least one queen
for i in range(n):
    clauses.append([varnum(i, j) for j in range(n)])

# each row contains at most one queen
for i in range(n):
    for j1, j2 in combinations(range(n), 2):
        clauses.append([-varnum(i, j1), -varnum(i, j2)])

# each column contains at most one queen
for j in range(n):
    for i1, i2 in combinations(range(n), 2):
        clauses.append([-varnum(i1, j), -varnum(i2, j)])

# no two queens stay on the same diagonal
for i1, j1, i2, j2 in product(range(n), repeat=4):
    if i1 == i2:
        continue

    if abs(i1 - i2) == abs(j1 - j2):
        clauses.append([-varnum(i1, j1),
                        -varnum(i2, j2)])

assignment = solve(clauses)
for i, j in product(range(n), repeat=2):
    if assignment[varnum(i, j) - 1] > 0:
        print(j, end=' ')