GEMS OF TCS

HEURISTIC ALGORITHMS

Sasha Golovnev
Semptermber 20, 2023
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.
Given a complete weighted graph, find a cycle (or a path) of minimum total weight (length) visiting each node exactly once.

length: 9
Nearest Neighbors

• Going to the nearest unvisited node at every iteration?
• Going to the nearest unvisited node at every iteration?
• Efficient, works reasonably well in practice
Nearest Neighbors

- Going to the nearest unvisited node at every iteration?
- Efficient, works reasonably well in practice
- May produce a cycle that is much worse than an optimal one
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
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:
• 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
Suboptimal Solution for Euclidean TSP

\[ \text{OPT} \approx 26.42 \]
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

\[ \text{OPT} \approx 26.42 \]
Suboptimal Solution for Euclidean TSP

\[ \text{OPT} \approx 26.42 \]
\[ \text{NN} \approx 28.33 \]
Suboptimal Solution for Euclidean TSP

OPT $\approx 26.42$

NN $\approx 28.33$

For Euclidean instances, the resulting cycle is $O(\log n)$-approximate
LOCAL SEARCH

• $s \leftarrow$ some initial solution
• $s \leftarrow$ 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

\begin{itemize}
  \item $s \leftarrow$ some initial solution
  \item while it is possible to change 2 edges in $s$ to get a better cycle $s'$:
    \begin{itemize}
      \item $s \leftarrow s'$
    \end{itemize}
  \item return $s$
\end{itemize}
Changing two edges in a suboptimal solution:
EXAMPLE

Changing two edges in a suboptimal solution:
Example

Changing two edges in a suboptimal solution:
Example

Changing two edges in a suboptimal solution:
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

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 $s$
• Computes a local optimum instead of a global optimum
Properties

• Computes a local optimum instead of a global optimum
• The larger $d$, the better the resulting solution and the higher is the running time
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
(\overline{x}_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor \overline{x}_2) \land (\overline{x}_1 \lor x_3) \land (x_2 \lor \overline{x}_3)
SAT

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

\[(x_1 \lor x_2 \lor x_3) \land (x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3) \land (x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3)\]
BACKTRACKING

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

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

\[x_1 = 0\]

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

\[x_1 = 0\]

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

\[x_2 = 0\]

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

\[x_3 = 0\]

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

\[x_1 = 0\]

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

\[x_2 = 0\]

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

\[x_3 = 0\]

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

\[x_4 = 0\]

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

\[x_1 = 0\]

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

\[x_2 = 0\]

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

\[x_3 = 0\]

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

\[x_4 = 0, 1\]

\[
\begin{align*}
() & \quad ()
\end{align*}
\]
EXAMPLE

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

\[x_1 = 0\]

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

\[x_2 = 0\]

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

\[x_3 = 0\]

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

\[x_4 = 0\]

\[()()\]

\[x_3 = 1\]

\[(\neg x_4)\]

\[x_4 = 1\]

\[()()\]
Example

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

\[x_1 = 0\]

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

\[x_2 = 0\]

\[x_3 = 0\]

\[x_4 = 0\]

\[x_4 = 1\]

\[x_2 = 1\]

\[x_3 = 1\]

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

\[()\]

\[x_4 = 0\]

\[x_4 = 1\]

\[()\]

\[()\]
Example

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

\[x_1 = 0\] \to \[\]

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

\[x_1 = 1\] \to \[(x_2 \lor \neg x_4)\]

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

\[x_2 = 1\] \to \[

\[x_3 = 0\] \to \[(x_4)(\neg x_4)\]

\[x_3 = 1\] \to \[(\neg x_4)\]

\[x_4 = 0\] \to \[

\[x_4 = 1\] \to \[

\]
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 **SolveSAT**(\(F[x \leftarrow 0]\)) = “sat”:
    return “sat”
BACKTRACKING ALGORITHM

• \textbf{SolveSAT}(F):
  • if $F$ has no clauses:
    return “sat”
  • if $F$ contains an empty clause:
    return “unsat”
  • \(x \leftarrow \text{unassigned variable of } F\)
  • if \(\text{SolveSAT}(F[x \leftarrow 0]) = \text{“sat”}\):
    return “sat”
  • if \(\text{SolveSAT}(F[x \leftarrow 1]) = \text{“sat”}\):
    return “sat”
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”
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
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
Applications
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
SAT HANDBOOK
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/
- 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

26 May 2016
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.
from pycosat import solve

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

print(solve(clauses))
print(solve(clauses[1:]))
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]
Is it possible to place $n$ queens on an $n \times n$ board such that no two of them attack each other?
Examples
Examples
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 queen is placed into cell $(i, j)$
- For $0 \leq i < n$, $i$th row contains $\geq 1$ queen:
  
  $(x_{i0} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1)$.
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: 
  $$(x_{i0} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1).$$

• For $0 \leq i < n$, $i$th row contains $\leq 1$ queen: 
  $$\forall 0 \leq j_1 \neq j_2 < n: (x_{ij_1} = 0 \text{ or } x_{ij_2} = 0).$$
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:
  \( (x_{i0} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1) \).
• 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:
  $$(x_{i0} = 1 \text{ or } x_{i2} = 1 \text{ or } \ldots \text{ or } x_{i(n-1)} = 1).$$
- 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=' ')