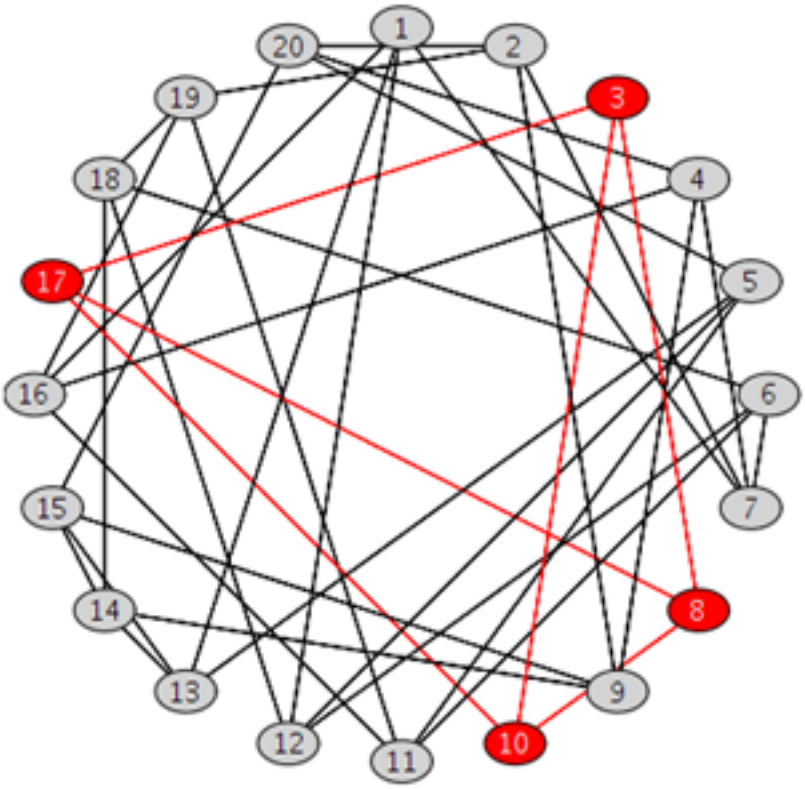
Clique Finding with SAT

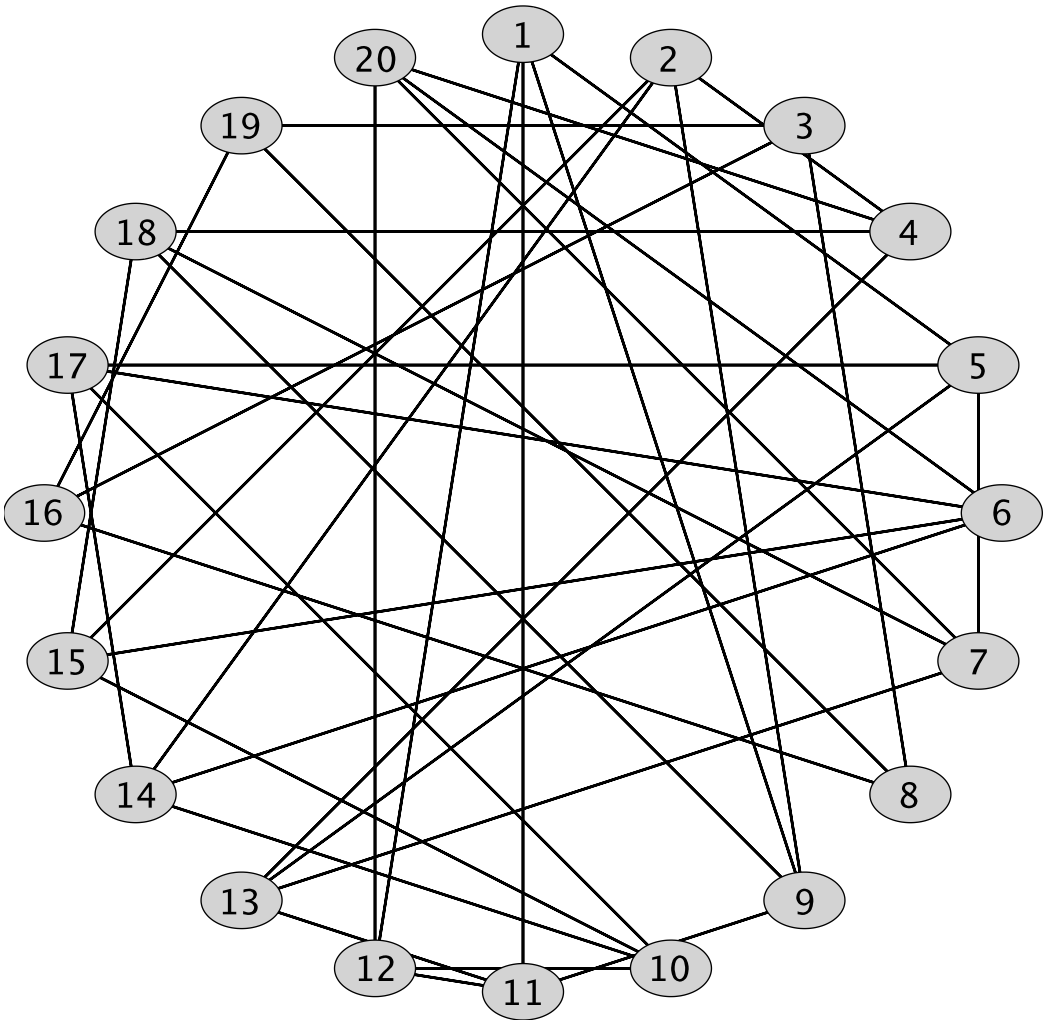
Curtis Bright, Maplesoft



- A *clique* of a graph is a subset of its vertices that are all mutually connected. The highlighted vertices in the above image show a clique of size 4.
- The following code generates a random graph G with n vertices such that every vertex has k neighbours and G contains a clique of size k.

```
n := 20:
k := 4:

with(GraphTheory):
with(RandomGraphs):
G := IsomorphicCopy(GraphUnion(RandomRegularGraph(n-k, k, connected), CompleteGraph([seq(n-k+i, i=1..k)])):
DrawGraph(G, style=circle);
```



- Although G contains a clique of size k it is not easy to find just by looking at the graph!
- In general, determining if a graph contains a clique of size k is an NP problem, meaning that it is easy to verify the correctness of a solution if

one can be found. It is also NP-complete, the hardest kind of problems in NP—and those for which no polynomial time algorithms are known.

- Although Maple can solve this problem using the FindClique function we can also solve this problem by encoding it in Boolean logic and using Maple's built-in efficient SAT solver. A SAT solver accepts a formula in Boolean logic and returns a satisfying assignment (if one exists). See the Satisfy command for more information.

Generating clique constraints

- We'll use the Boolean variables x_i where i is a vertex of G to represent if the vertex i appears in the clique of size k that we are attempting to find.
- We need to enforce a constraint that says that if x_i and x_j are true (for any vertices $1 \le i, j \le n$) then the edge $\{i, j\}$ exists in the graph G.
- Equivalently, if the edge $\{i, j\}$ does *not* exist in the graph G then the variables x_i and x_j cannot both be true (for any vertices i, j). In other words, if $\{i, j\}$ is an edge of the complement of G then we know the clause $\neg x_i \vee \neg x_j$.

```
complementEdges := Edges(GraphComplement(G)):
cliqueConstraints := Array(1..nops(complementEdges)):

for l from 1 to nops(complementEdges) do
     i, j := complementEdges[l][];
     cliqueConstraints[l] := &not(x[i]) &or &not(x[j]
);
end do:
cliqueConstraints := entries(cliqueConstraints,
nolist):
```

Generating size constraints

- Additionally, we need a way to enforce that the found clique is of size k. The most naive way to encode this is $\left(x_1 \wedge \cdots \wedge x_k\right) \vee \cdots \vee \left(x_{n-k+1} \wedge \cdots \wedge x_n\right)$ but this encoding is very inefficient in practice.

- A cleverer encoding uses Boolean counter variables $s_{i,j}$ (where $0 \le i \le n$ and $0 \le j \le k$) that represent that at least j of the variables x_1, ..., x_i are assigned to true.
- We know that $s_{0,j}$ will be false for $1 \le j \le k$ and that $s_{i,0}$ will be true for $0 \le i \le n$.
- Additionally, we know that $s_{i,j}$ is true if and only if $s_{i-1,j}$ is true or x_i is true and $s_{i-1,j-1}$ is true. This is represented by the clauses $s_{i,j} \Leftrightarrow \left(s_{i-1,j} \vee \left(x_i \wedge s_{i-1,j-1}\right)\right)$ for $1 \le i \le n$ and $1 \le j \le k$.
- To enforce that the found clique contains at least k vertices we also assign $s_{n,k}$ to true.

```
1  baseConstraints := seq(s[i, 0], i=0..n), seq(&not(s[0
   , j]), j=1..k):
2
3  inductiveConstraints := Array(1..n, 1..k, (i, j) -> s
   [i, j] &iff (s[i-1, j] &or (x[i] &and s[i-1, j-1]))):
4  inductiveConstraints := entries(inductiveConstraints,
   nolist):
5
6  sizeConstraint := s[n, k]:
```

Finding a *k*-vertex clique

- We use the Satisfy command from the Logic package which finds a satisfying assignment of a logical formula if one exists.
- We use the Usage command from the CodeTools package to measure how quickly the solution is found.

```
allConstraints := cliqueConstraints, baseConstraints,
inductiveConstraints, sizeConstraint:

satisfyingAssignment := CodeTools:-Usage(Logic:-
Satisfy(&and(allConstraints))):
```

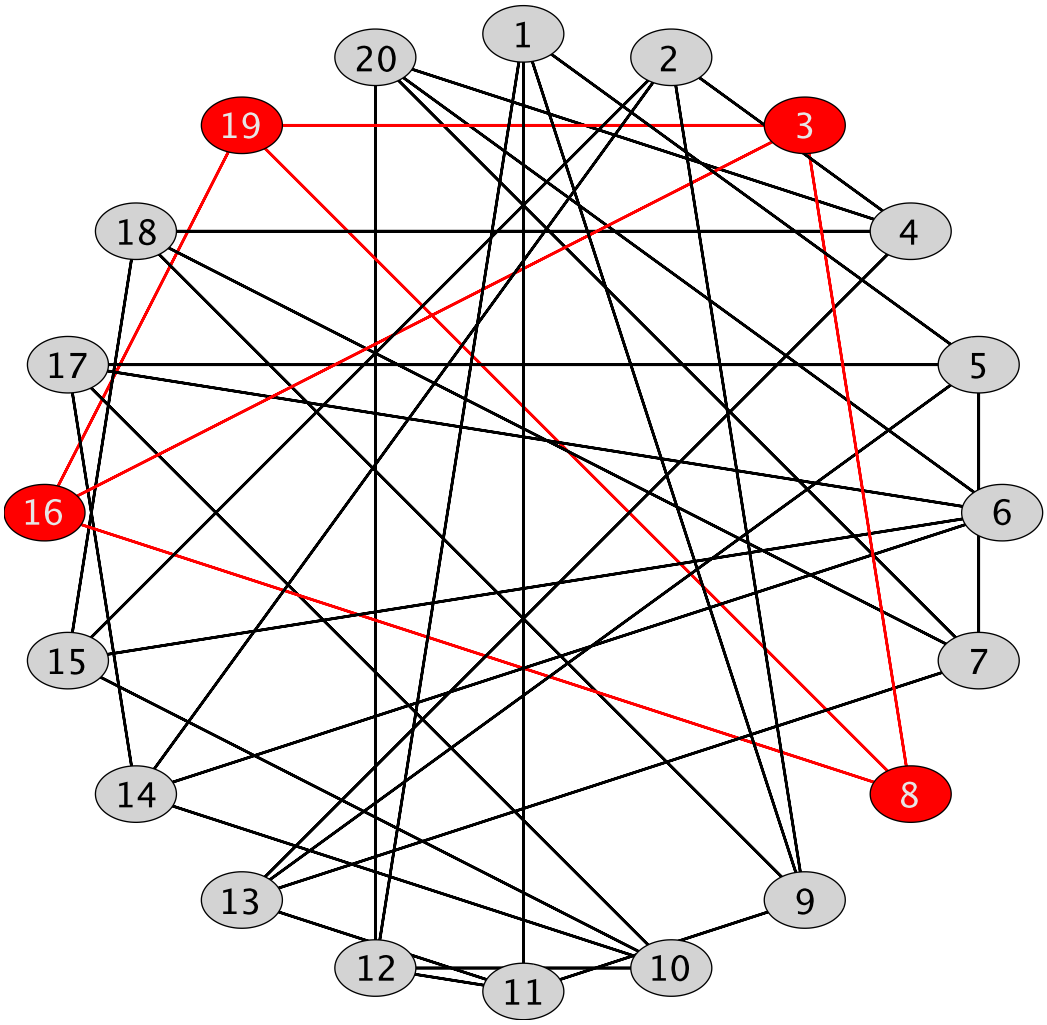
```
memory used=11.33MiB, alloc change=33.00MiB, cpu time=156.00ms,
real time=388.00ms, gc time=46.80ms
```

Visualizing the solution

- To display the found clique we find the variables x_i that have been assigned to true in the satisfying assignment (only using the first k if more than k have been assigned to true).
- We visualize the solution by highlighting the vertices of the found clique and the edges between them.

```
cliqueAssignments := select(type,
satisfyingAssignment, specindex(x) = identical(true))
:
clique := map2(op, [1, 1], cliqueAssignments[1..k]):

HighlightVertex(G, clique, red):
HighlightEdges(G, [seq(seq({clique[i], clique[j]}, j=
i+1..k), i=1..k)], [seq(red, i=1..k*(k-1)/2)]):
DrawGraph(G, style=circle);
```



Solving challenge instances

- Some difficult clique finding problems can be quickly solved using the above reduction to SAT. The following challenge instance is from the Second DIMACS Implementation Challenge.

```
# Graph MANN_a9 in Graph6 format
MANN_a9 :=
"l~~~~~~^|~Z{z~|~^^fz~~nz~^b~~~n~z~^{^~~~^~~^v~{^~~~
~~~^n~~d~~~~z~~~z~^~~d~~~~~n~~~z|~~~{n~~~~~v~~~~^v~~
{v~~~~~}~~~~~^z~~~~e~~~~~~|~~~~~z}~~~~~e~~~~~~~^~~~~
z|~~~~~{":
G := Import(MANN_a9, format="Graph6", source=direct);
```

G := Graph 1: an undirected unweighted graph with 45 vertices and 918 edge(s)

- We can generate the same set of constraints for this graph:

```
n := NumberOfVertices(G):
complementEdges := Edges(GraphComplement(G)):
cliqueConstraints := Array(1..nops(complementEdges)):
for l from 1 to nops(complementEdges) do
     i, j := complementEdges[l][];
     cliqueConstraints[l] := &not(x[i]) &or &not(x[j]);
end do:
cliqueConstraints := entries(cliqueConstraints, nolist)
:
baseConstraints := seq(s[i, 0], i=0..n), seq(&not(s[0,
j]), j=1..n):
inductiveConstraintsA := Array(1..n, 1..n, (i, j) -> s[
i, j] &iff (s[i-1, j] &or (x[i] &and s[i-1, j-1]))):
```

- We now call the SAT solver with increasingly large values of k until a clique of size k does not exist in G. For efficiency reasons, we only pass the inductive constraints necessary to define $s_{n,k}$ for the current value of k. Maple 2018 required minutes to find the largest clique in this instance, but the SAT approach can find it in seconds.

```
start := time[real]():
satisfyingAssignment := 'satisfyingAssignment':
for k from 1 to n while satisfyingAssignment <> NULL
do
     inductiveConstraints := entries(
inductiveConstraintsA[1..n, 1..k], nolist):
     sizeConstraint := s[n, k]:
     allConstraints := cliqueConstraints,
baseConstraints, inductiveConstraints, sizeConstraint
:
     satisfyingAssignment := Logic:-Satisfy(&and(
allConstraints)):
end do:
printf(
"The largest clique of G has size %d and it was foun
 in %.2f seconds.\n", k-2, time[real]()-start);
```

```
The largest clique of G has size 16 and it was found in 6.79
seconds.
```