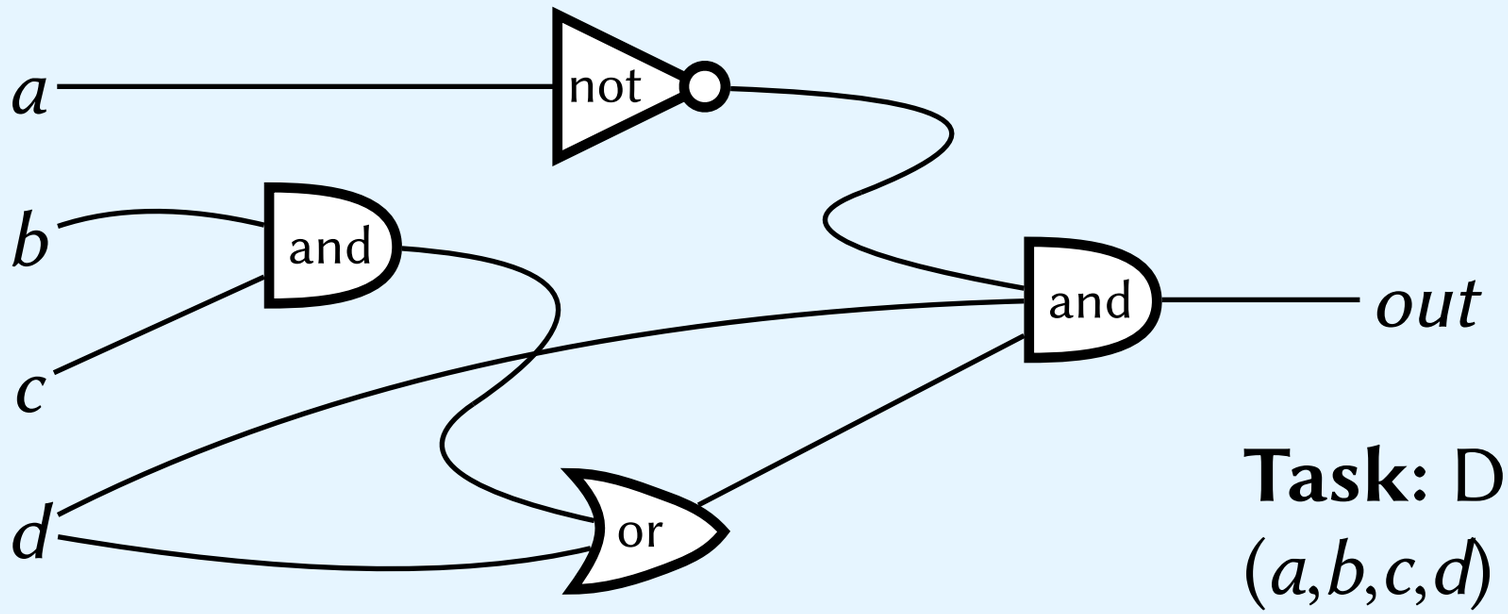
Announcements

- All coursework repositories now created!
- Lab exercise 2 due in a couple of hours!
- Guest lecture next Thursday!

A difficult problem



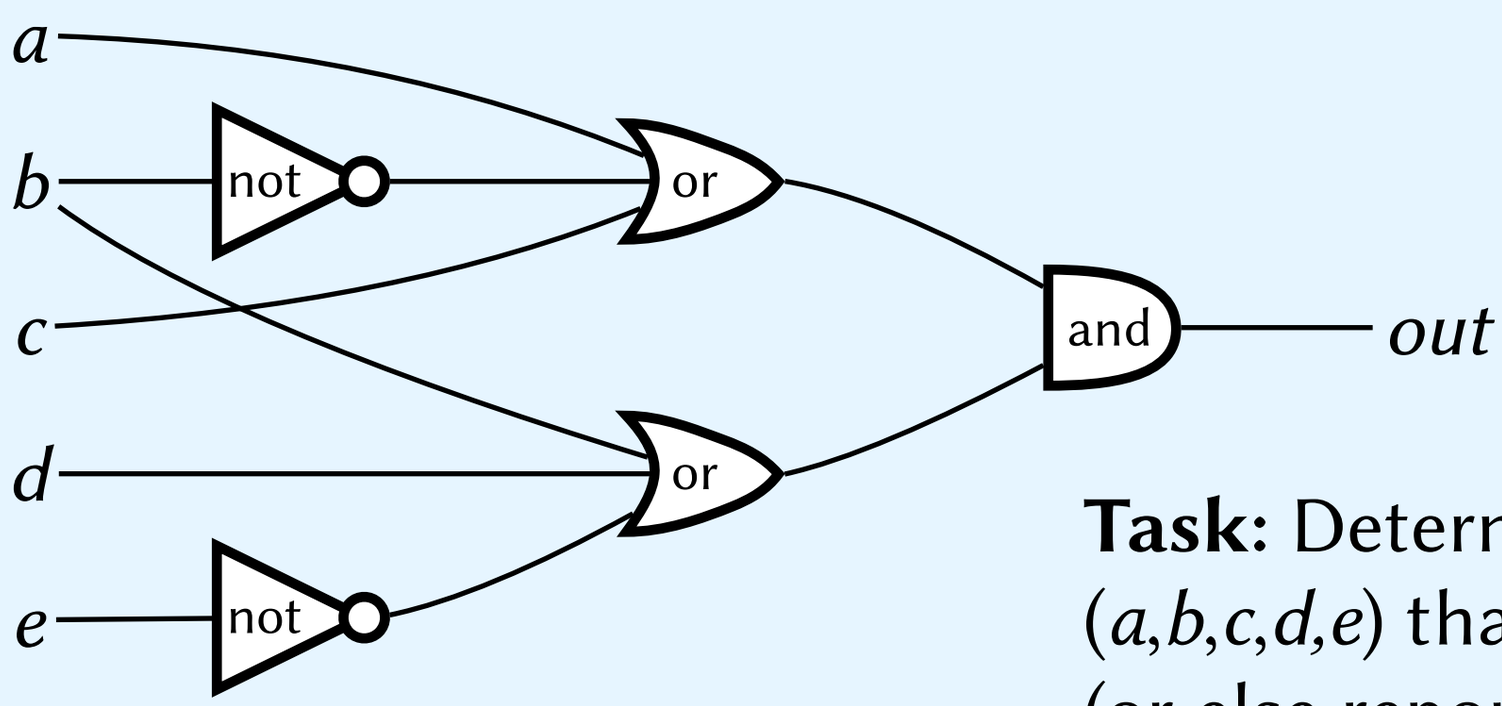
Task: Determine values for (a,b,c,d) that make *out* high (or else report "impossible")

- Possible algorithm: try all 2^4 possible input vectors. But this is exponential in the number of inputs!

- Nobody has come up with a faster algorithm.

A difficult problem

- The problem is *just as hard* if we restrict the form of circuits to be a layer of **NOT**-gates, followed by a layer of 3-input **OR**-gates, followed by a single *n*-input **AND**-gate.



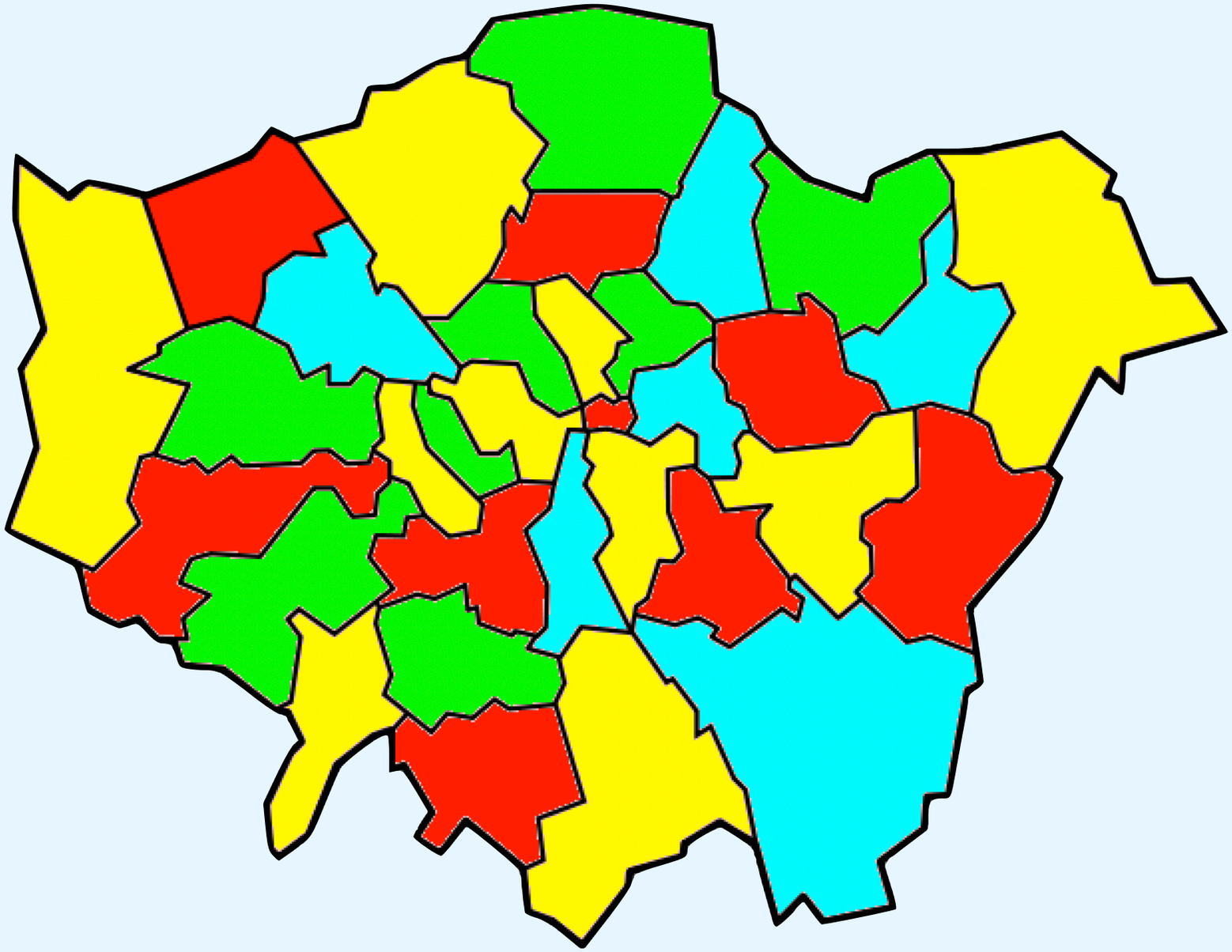
Task: Determine values for (*a*,*b*,*c*,*d*,*e*) that make *out* high (or else report "impossible")

- This problem is called "3SAT".

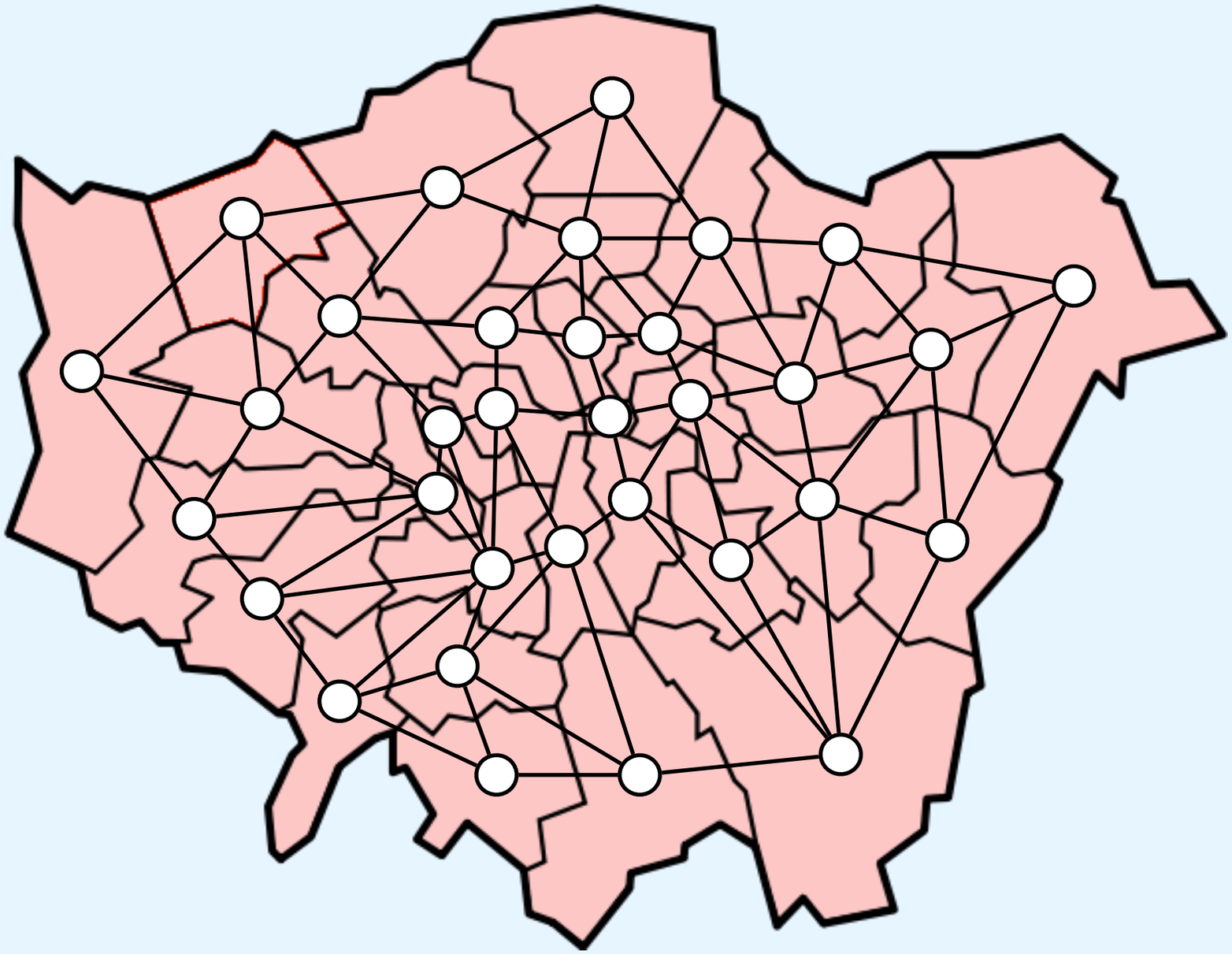
Colouring maps

you are here

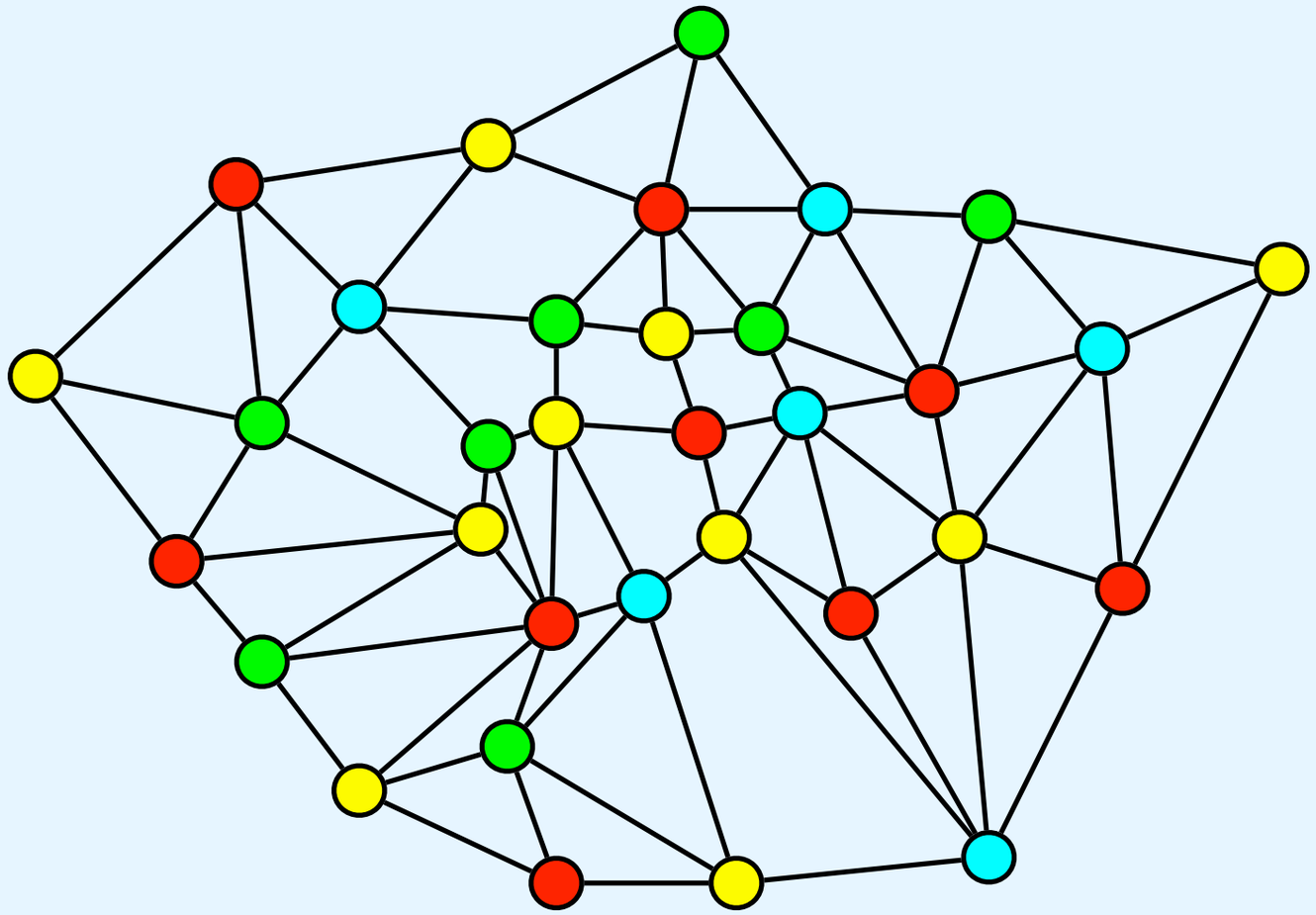
Colouring maps



Colouring graphs

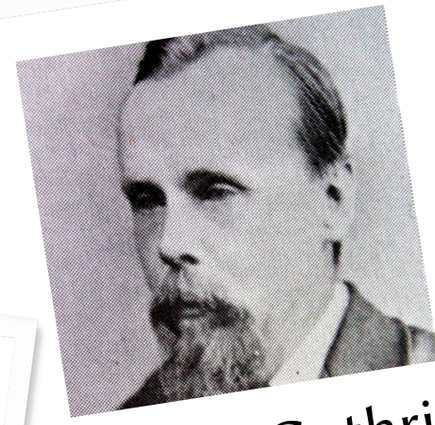


Colouring graphs



Colouring graphs

- In 1852, Francis Guthrie conjectured that four colours were enough for a *planar* graph.



Francis Guthrie
1831–1899

Kenneth Appel
1932–2013

- In 1977, Kenneth Appel and Wolfgang Haken proved (with the help of a computer program) that he was right!

Wolfgang Haken
1928–2022

> What reason is there for saying that the 4CT is not really a theorem or that mathematicians have not really produced a proof of it? Just this: no mathematician has seen a proof of the 4CT, nor has any seen a proof that it has a proof. Moreover, it is very unlikely that any mathematician will ever see a proof of the 4CT.

- In 2005, Georges Gonthier mechanised their proof in Coq.

Georges Gonthier

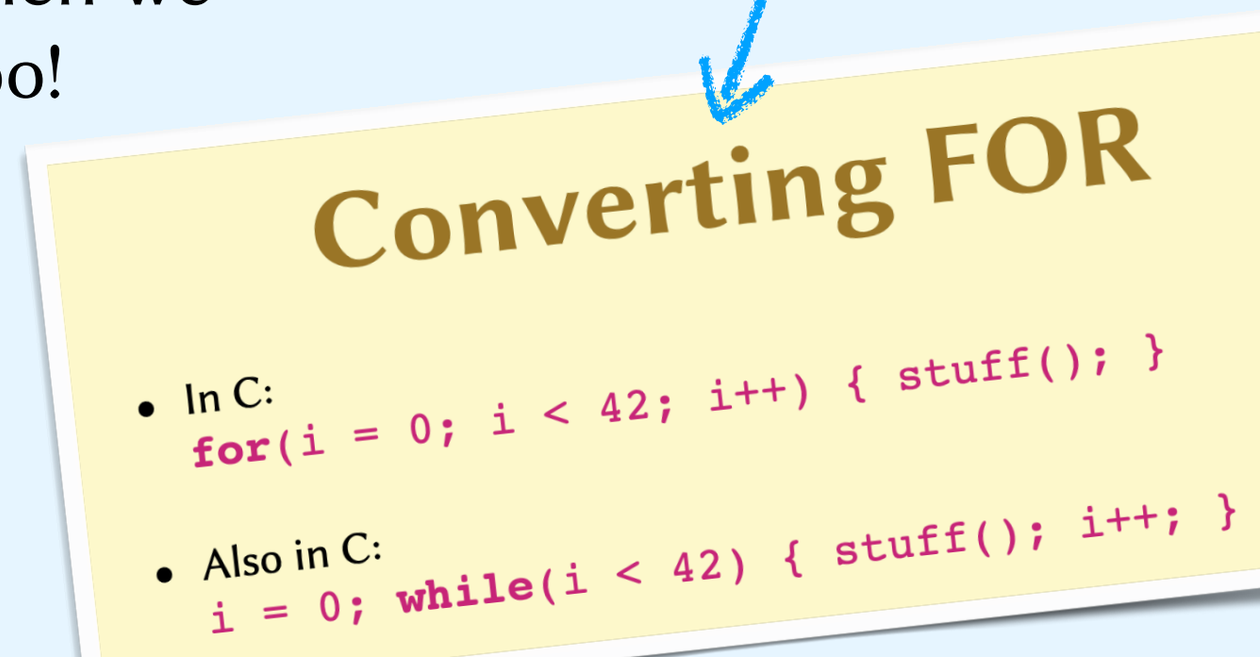
3-colourability

- **Task:** given a graph, determine a colour (**red**, **green**, or **blue**) for each vertex so that no connected vertices have the same colour (or else report "impossible").

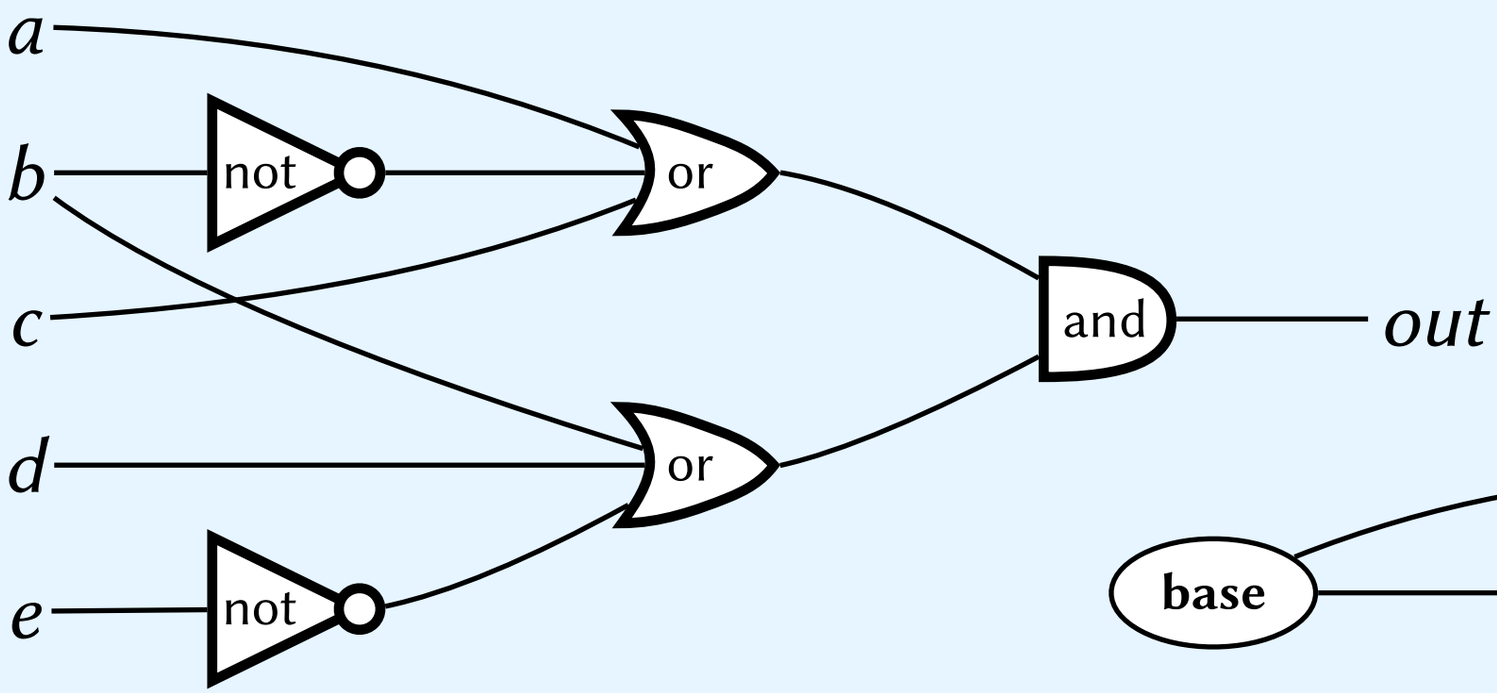
- This is difficult (exponential in the number of vertices) because if we could solve *this* problem efficiently, then we could solve "3SAT" efficiently too!

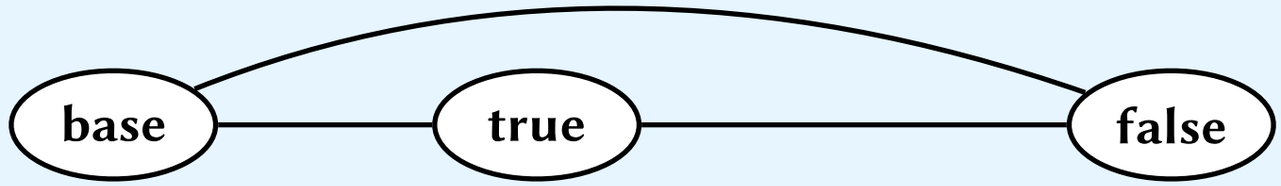
- We say that 3SAT **reduces to** 3-colourability.

The problem of compiling FOR loops reduces to the problem of compiling WHILE loops.

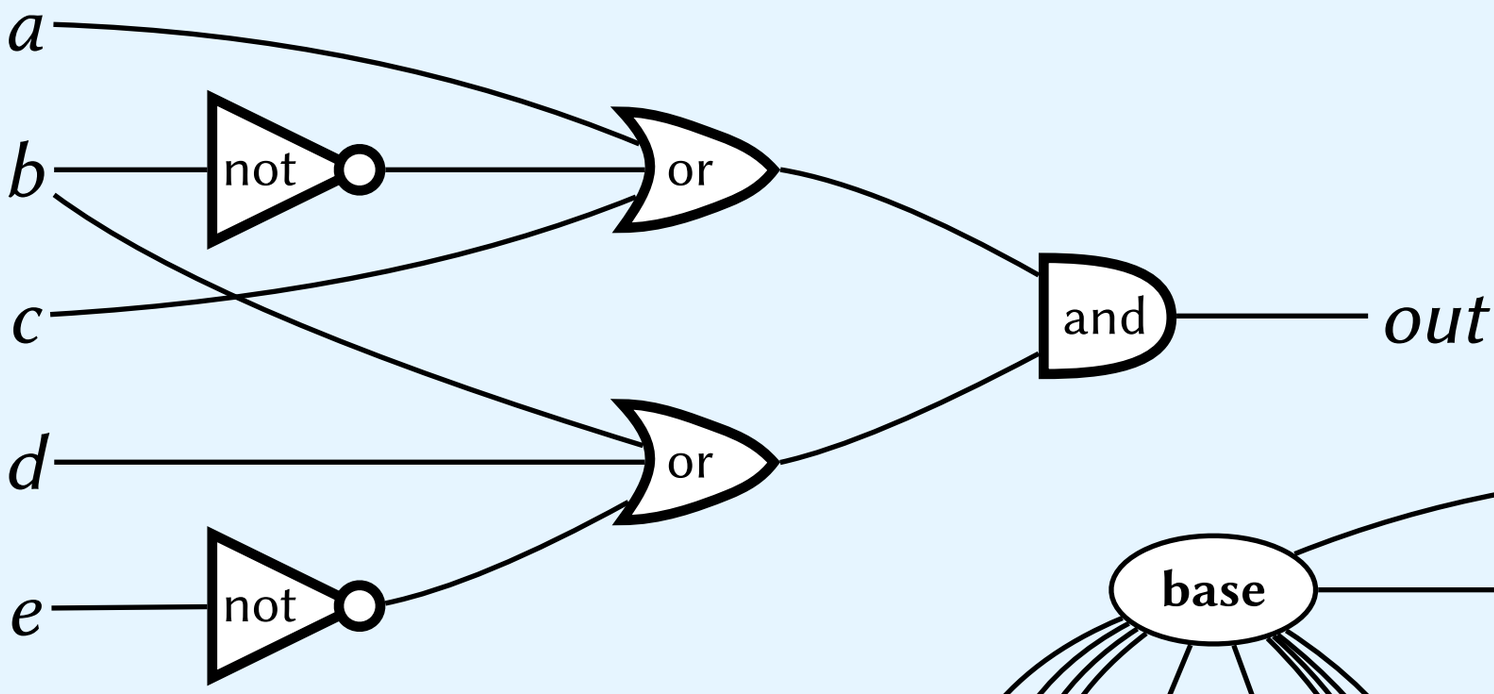


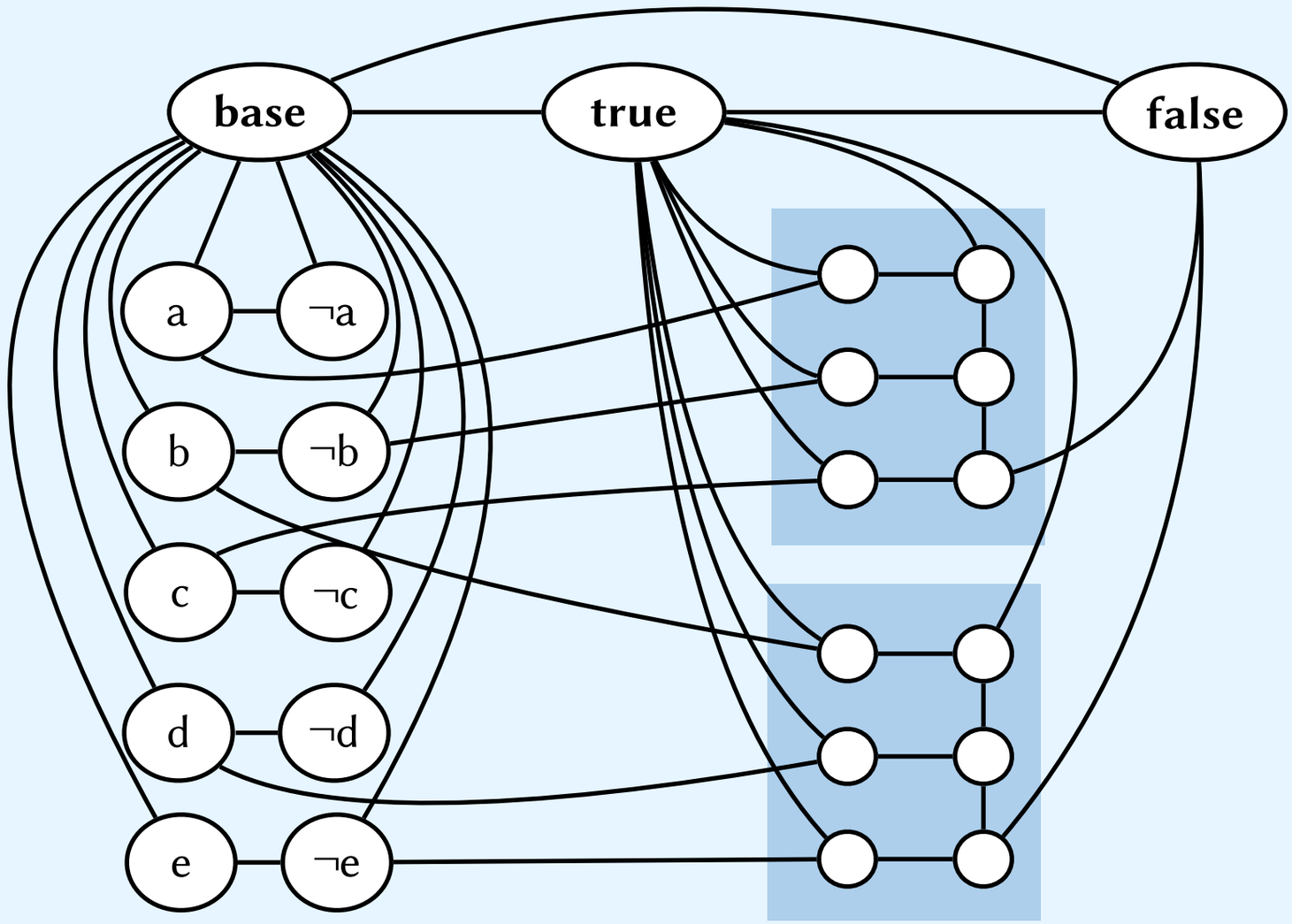
From 3SAT to 3-colourability



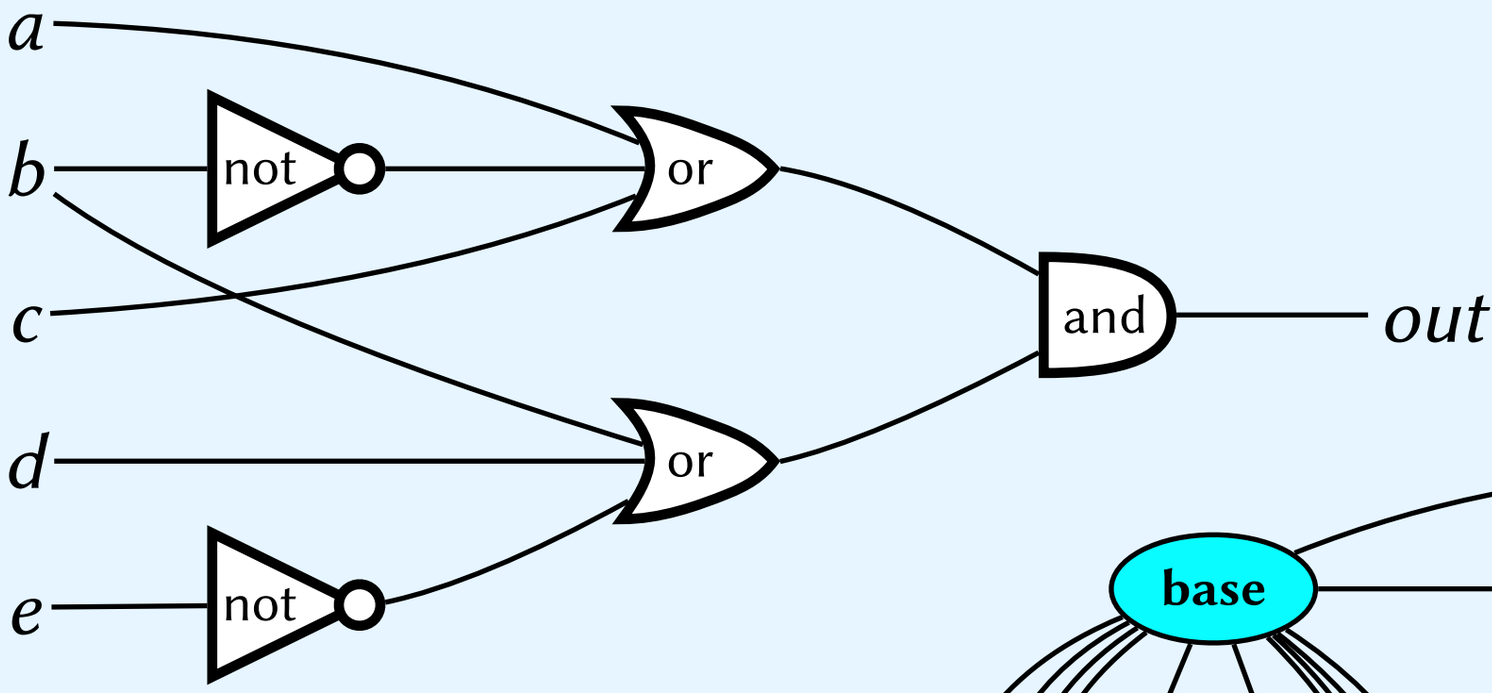


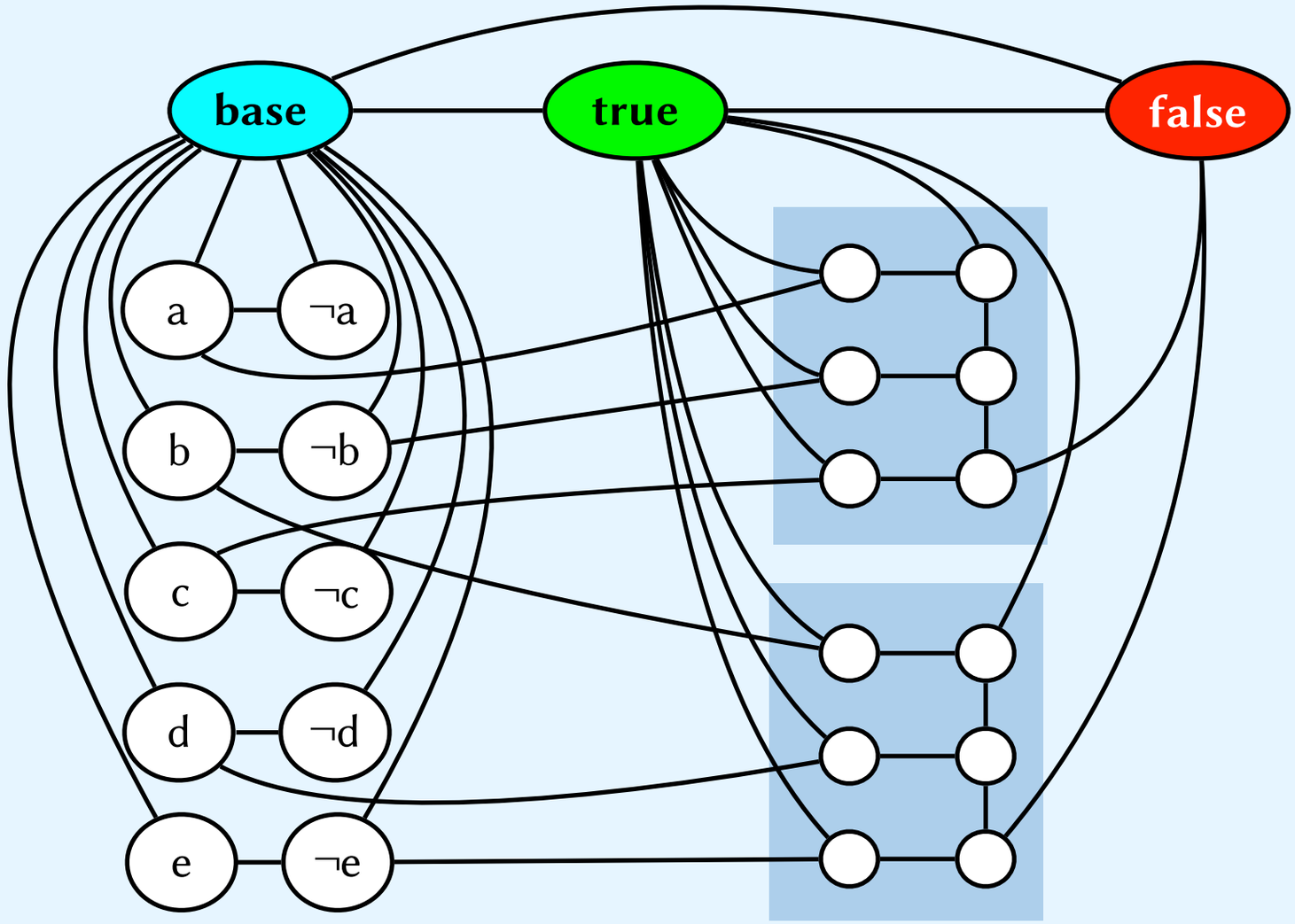
From 3SAT to 3-colourability



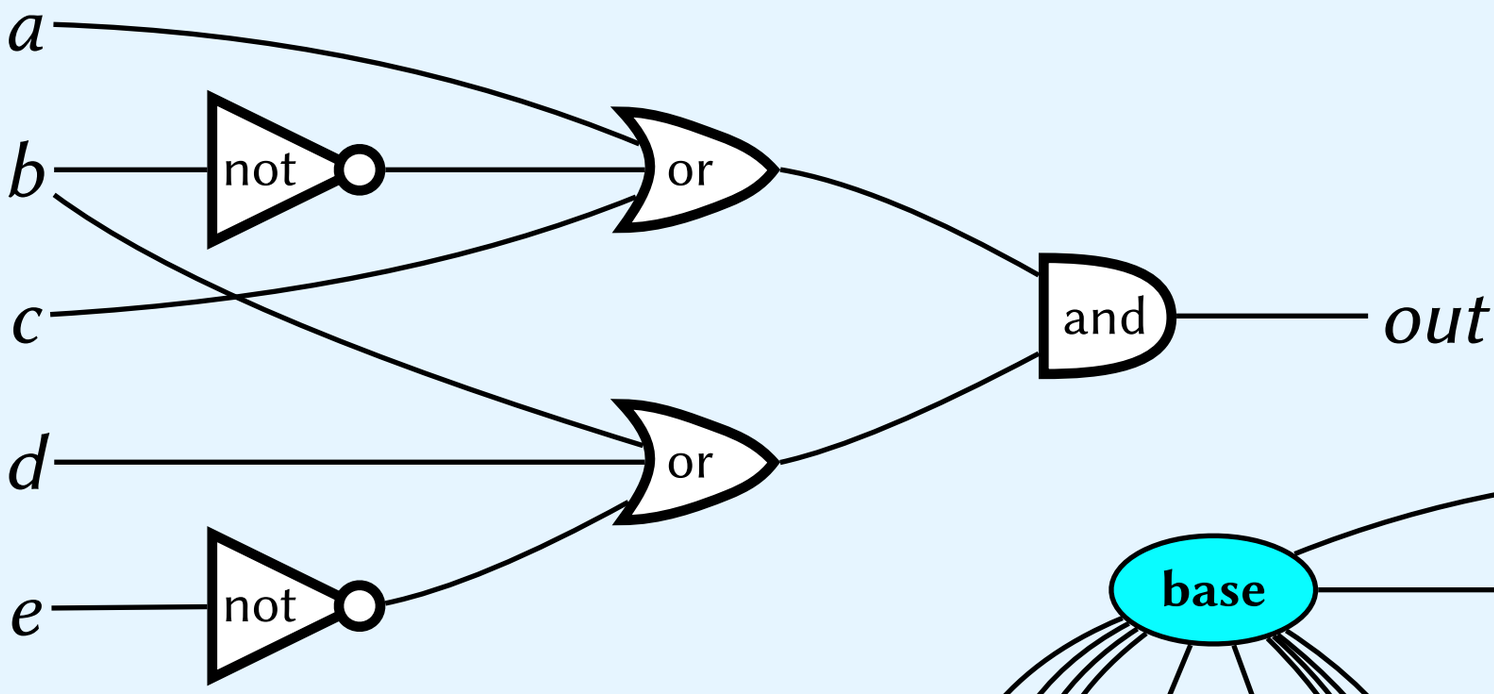


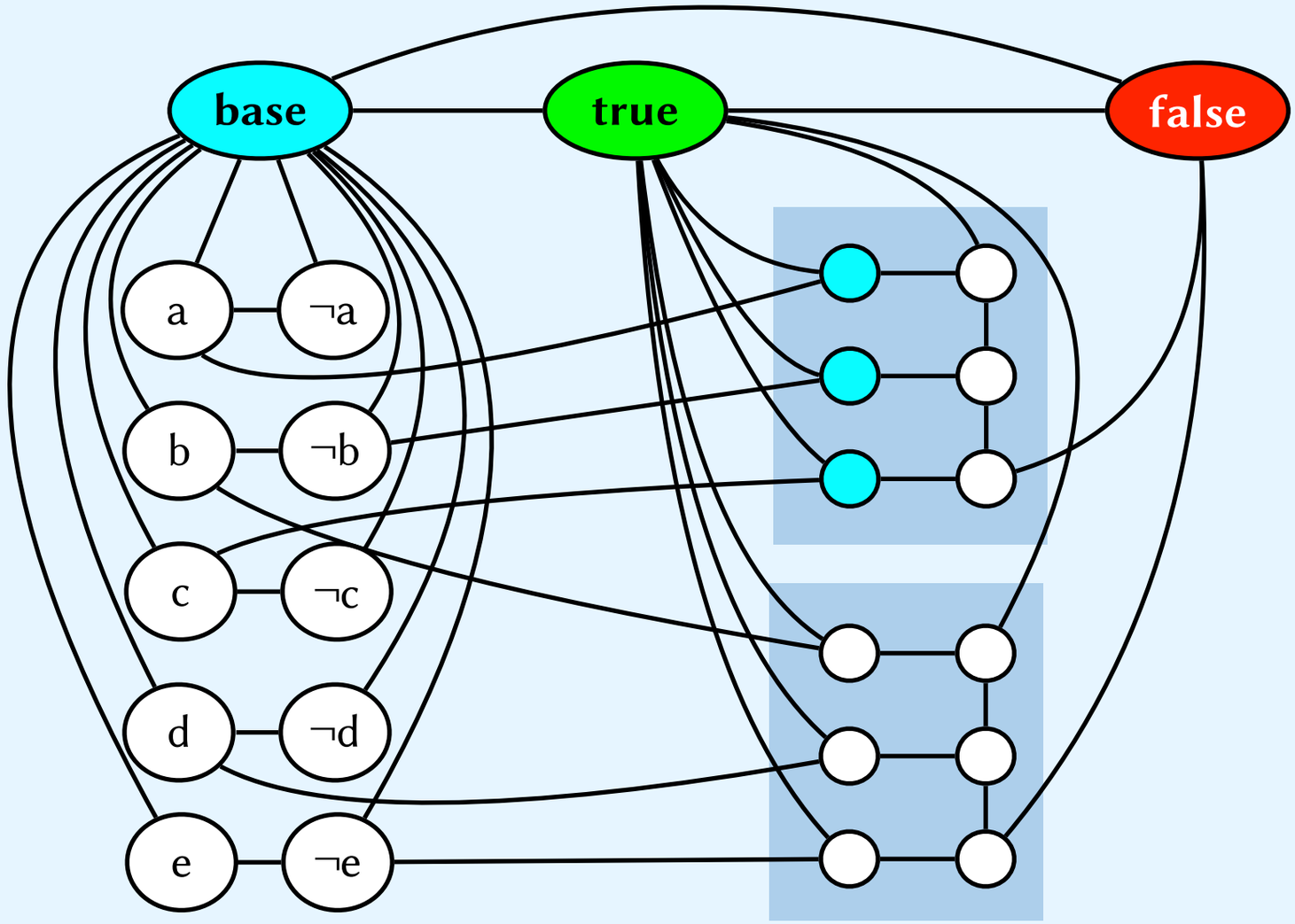
From 3SAT to 3-colourability



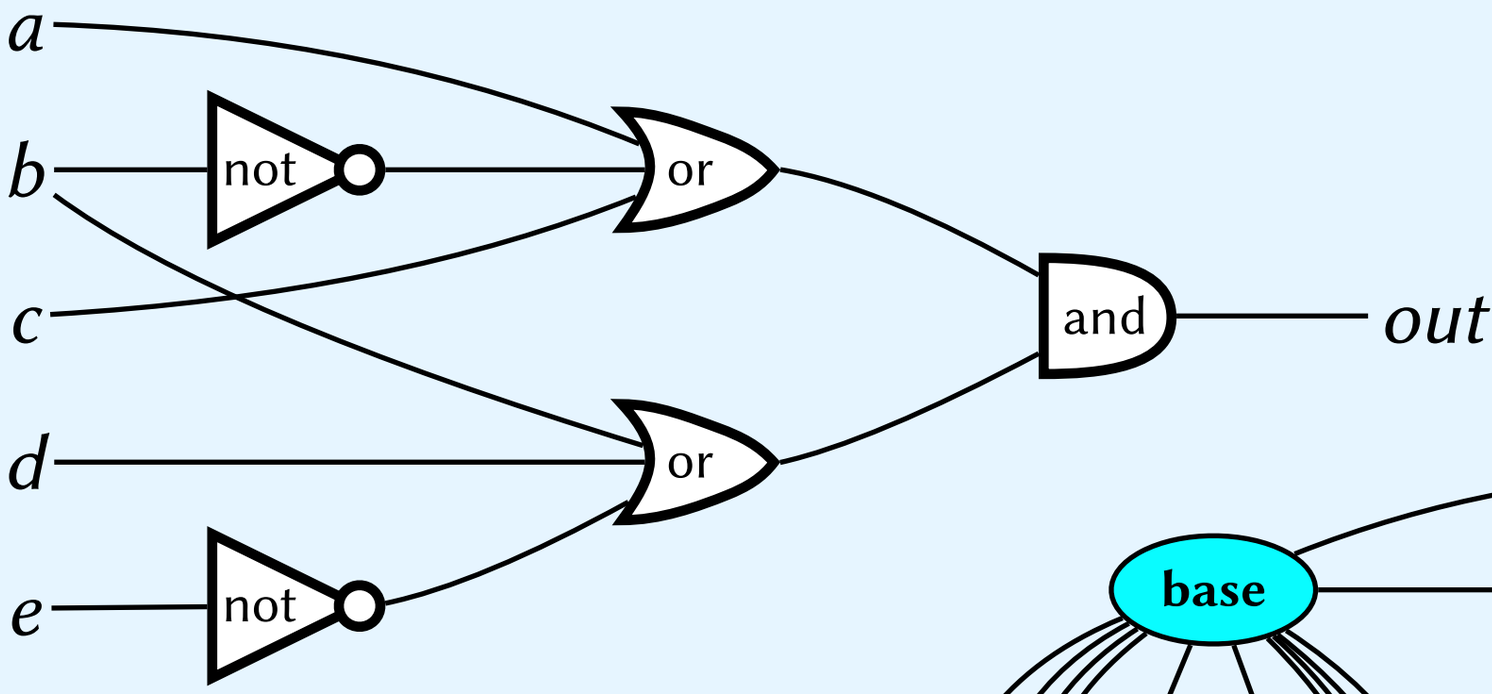


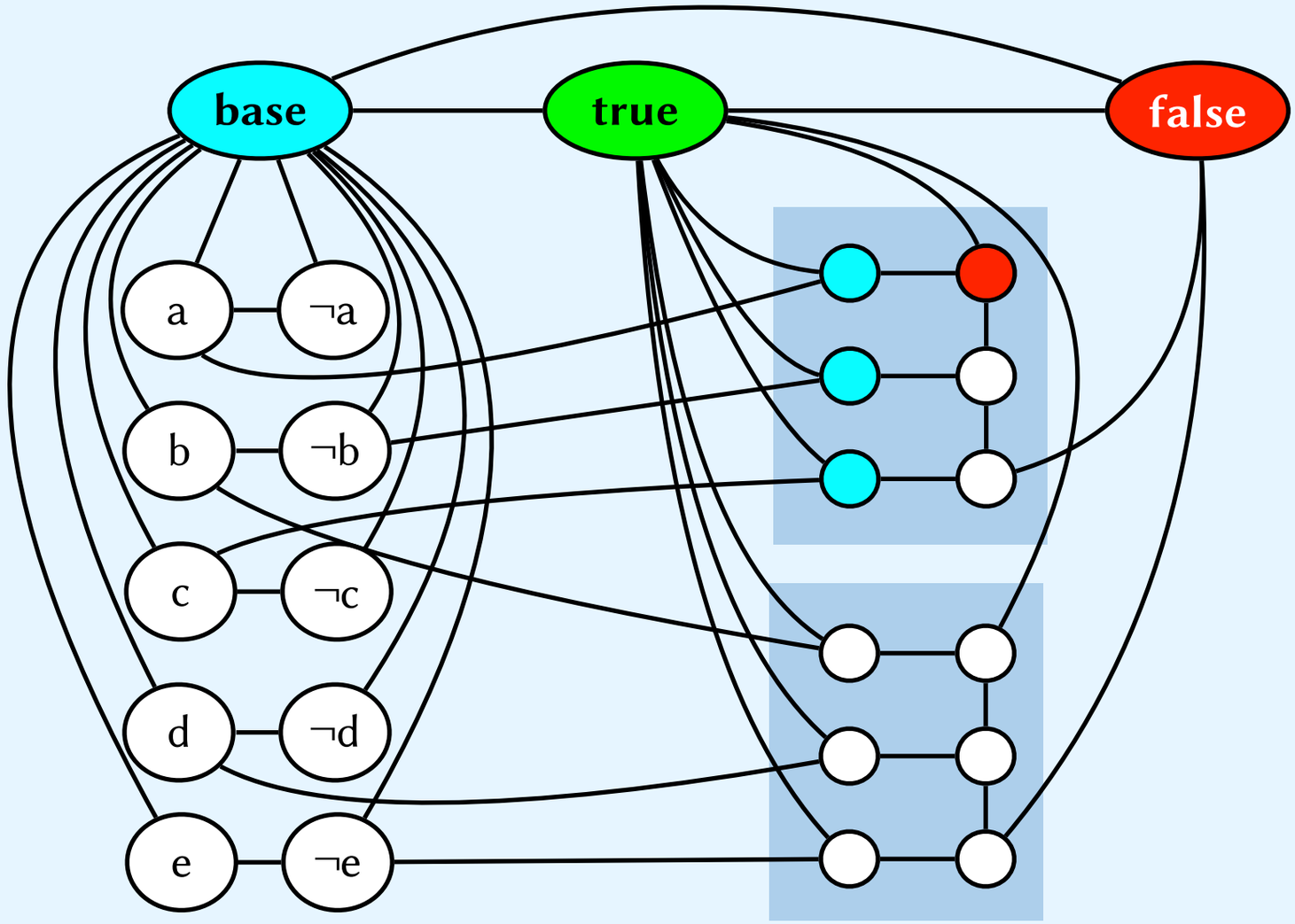
From 3SAT to 3-colourability



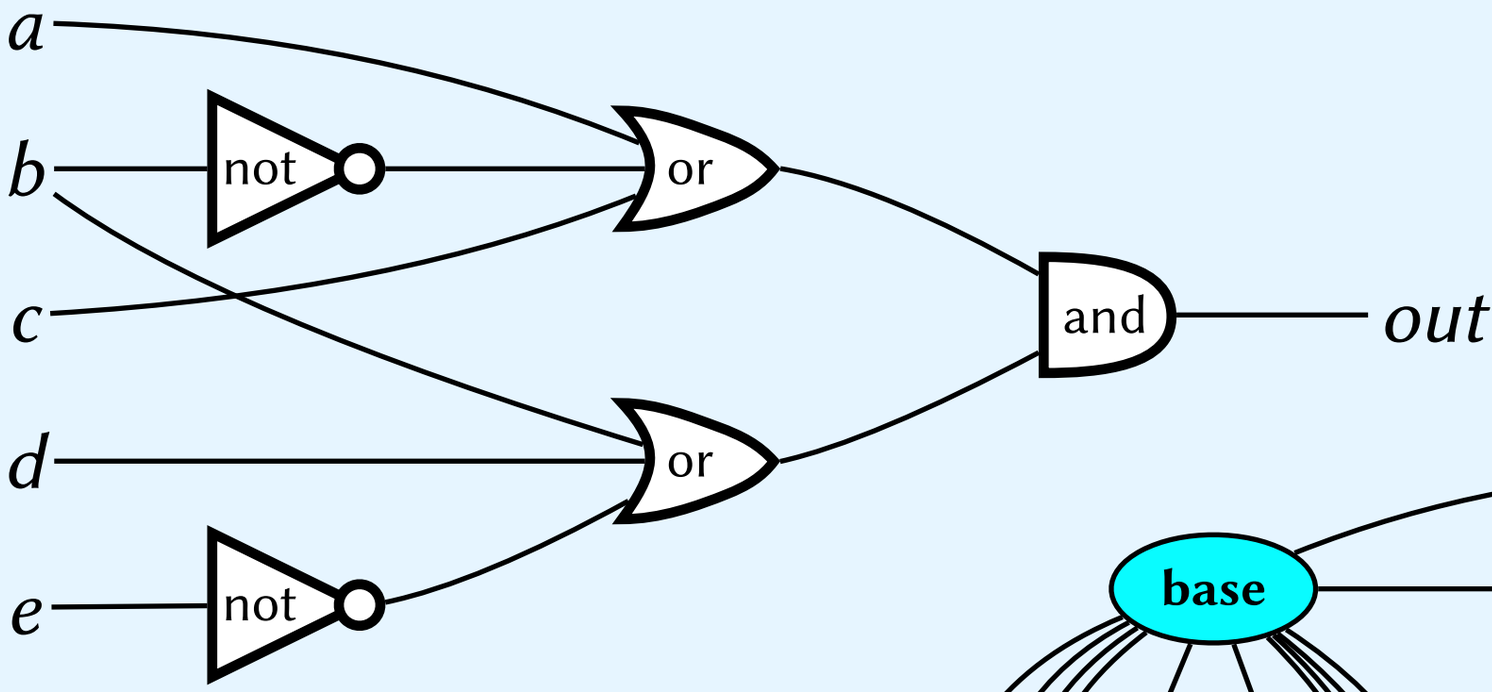


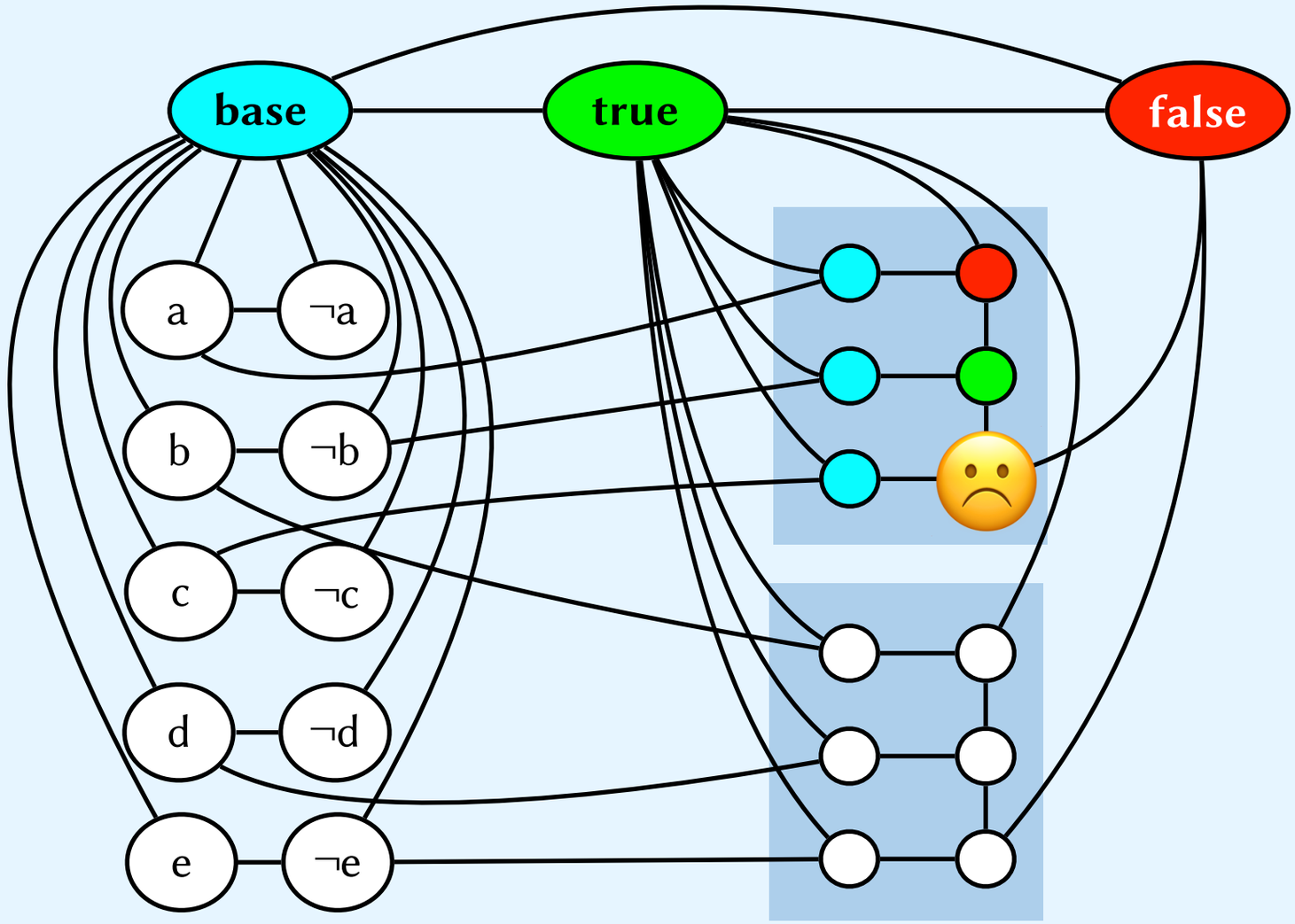
From 3SAT to 3-colourability



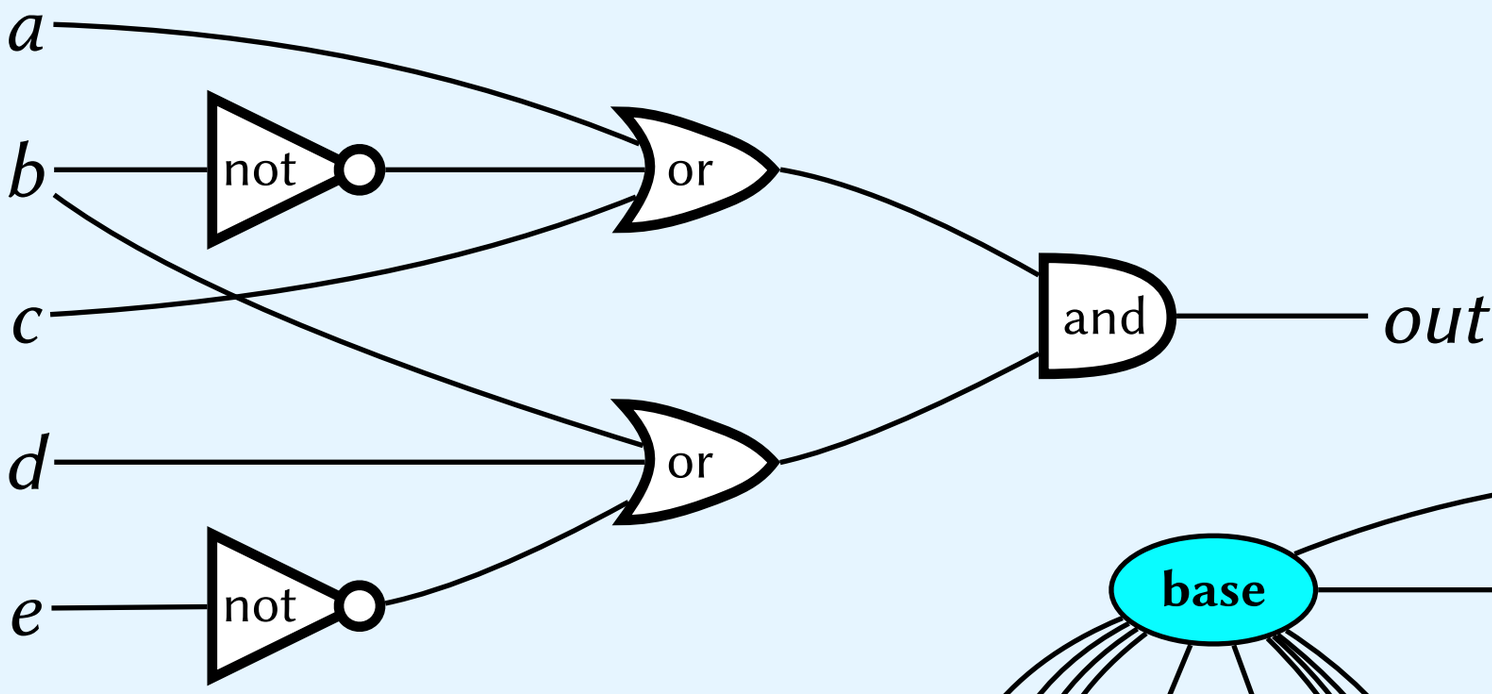


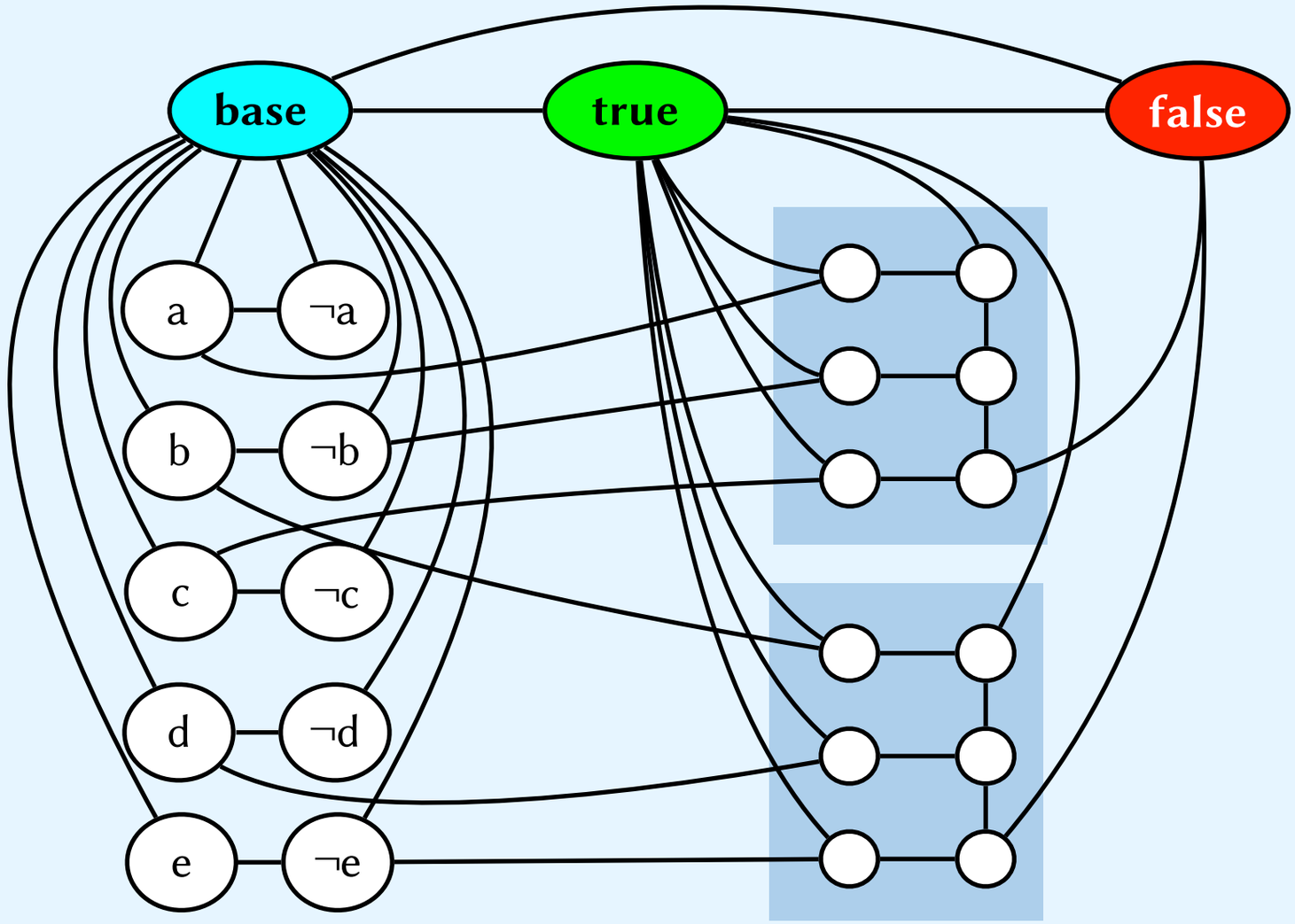
From 3SAT to 3-colourability



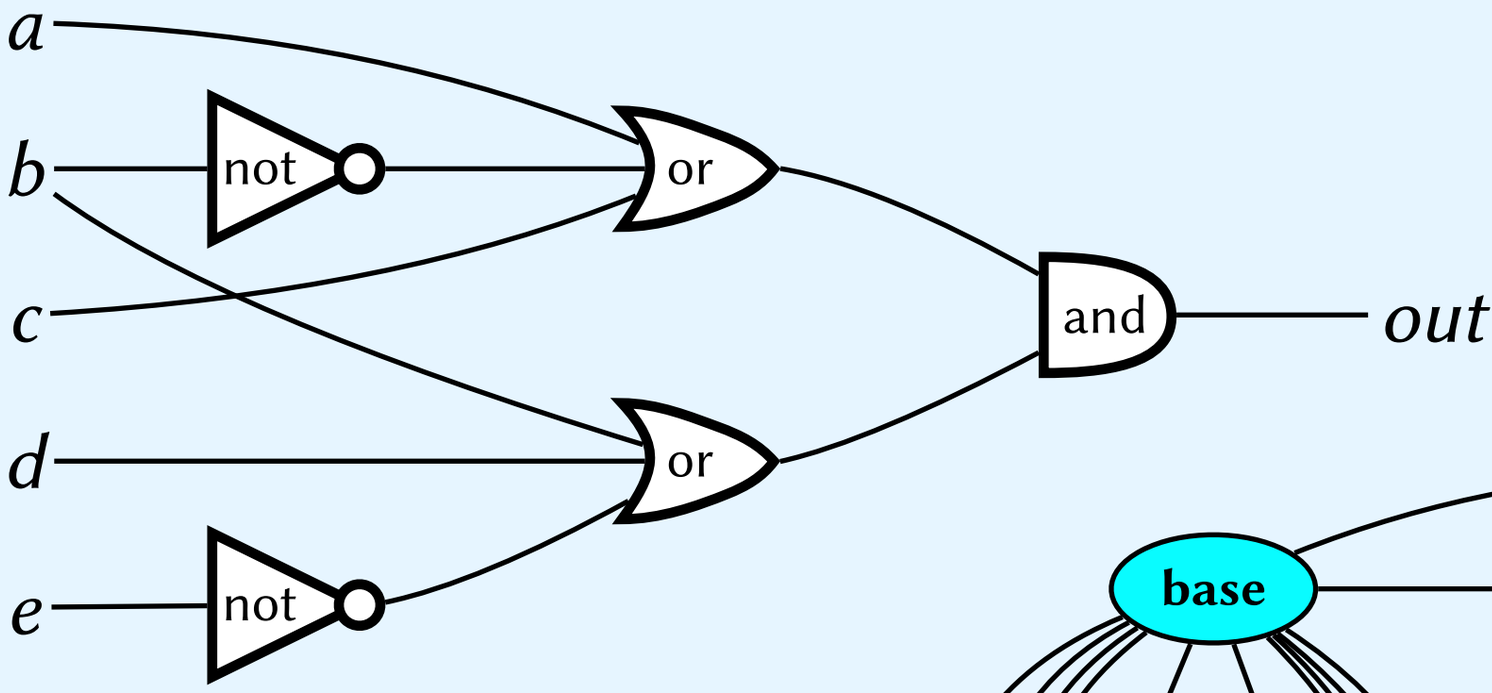


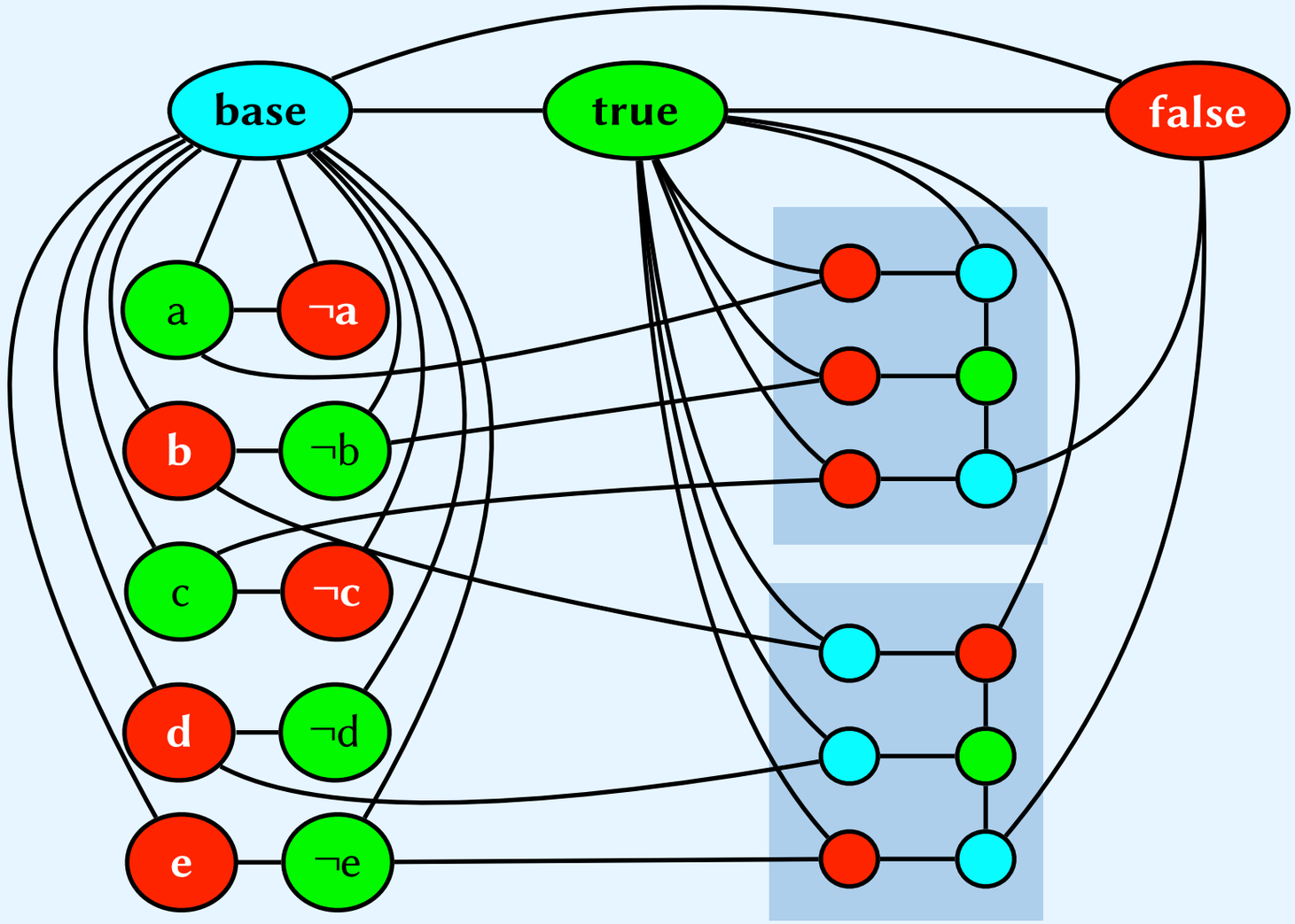
From 3SAT to 3-colourability





From 3SAT to 3-colourability





Summary

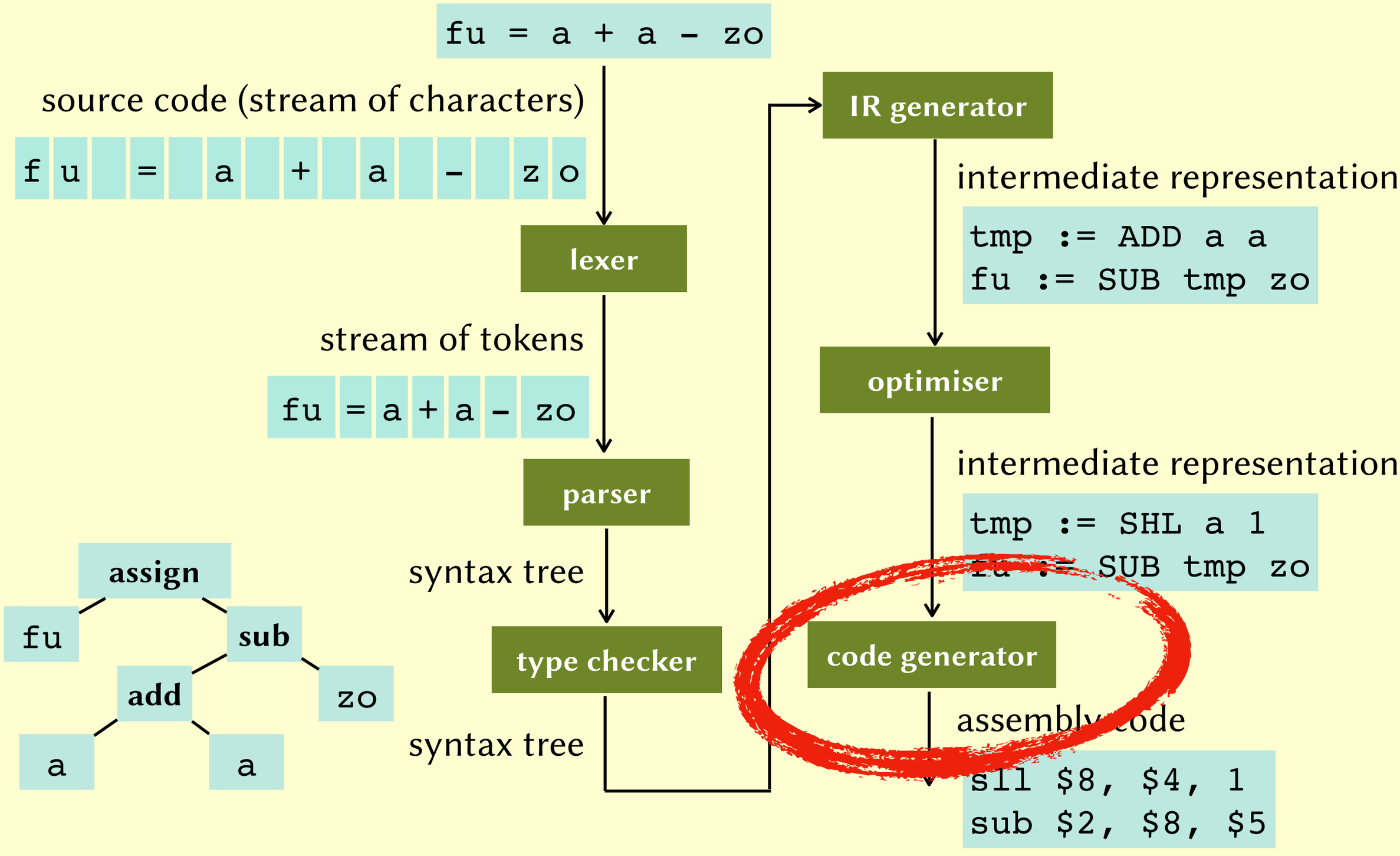
- SAT is a **hard problem**.
- 3SAT is **just as hard**.
- 3SAT can be reduced to **graph-colouring**.
- So graph-colouring is also a **hard problem**.

Lecture 8:
Code Generation

John Wickerson

Compilers

Anatomy of a compiler



Code generation

- Pick an appropriate machine instruction (or sequence of machine instructions) for each IR statement.
- The register allocation problem: decide how to make the best use of a **limited set of registers**.

Code generation

- In 3AC:

```
a = b + c;
```

- In machine code:

```
LW  t0, 0(b)
LW  t1, 0(c)
ADD t0, t0, t1
SW  t0, 0(a)
```

Code generation

- In 3AC:

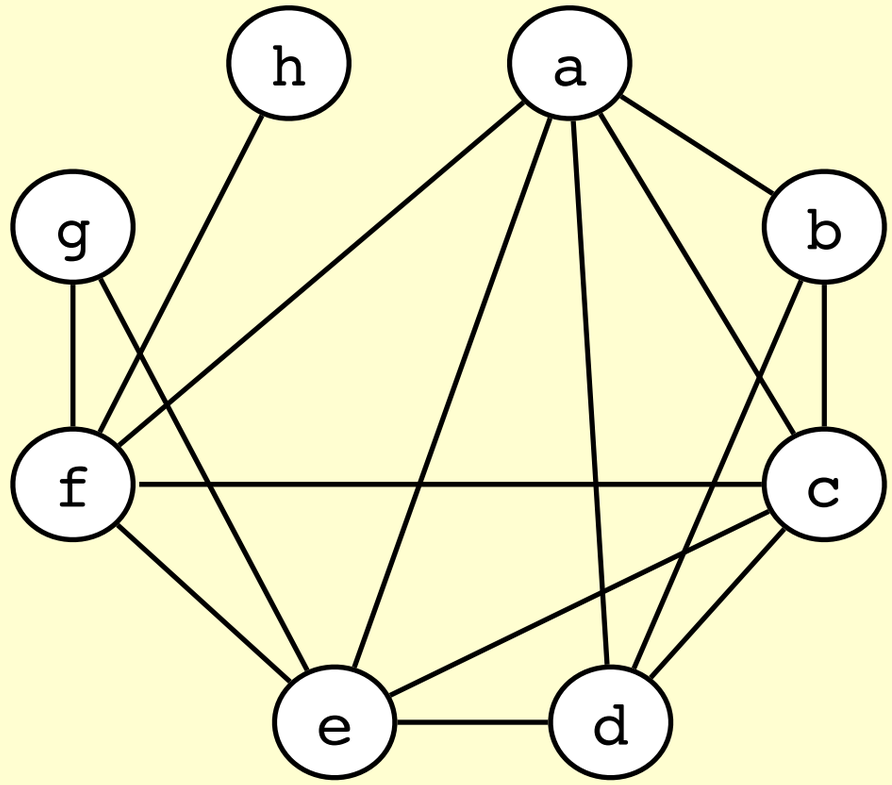
```
a = b + c;
d = a + e;
```

- In machine code:

```
LW  t0, 0(b)
LW  t1, 0(c)
ADD t0, t0, t1
SW  t0, 0(a)
LW  t0, 0(a)
LW  t1, 0(e)
ADD t0, t0, t1
SW  t0, 0(d)
```

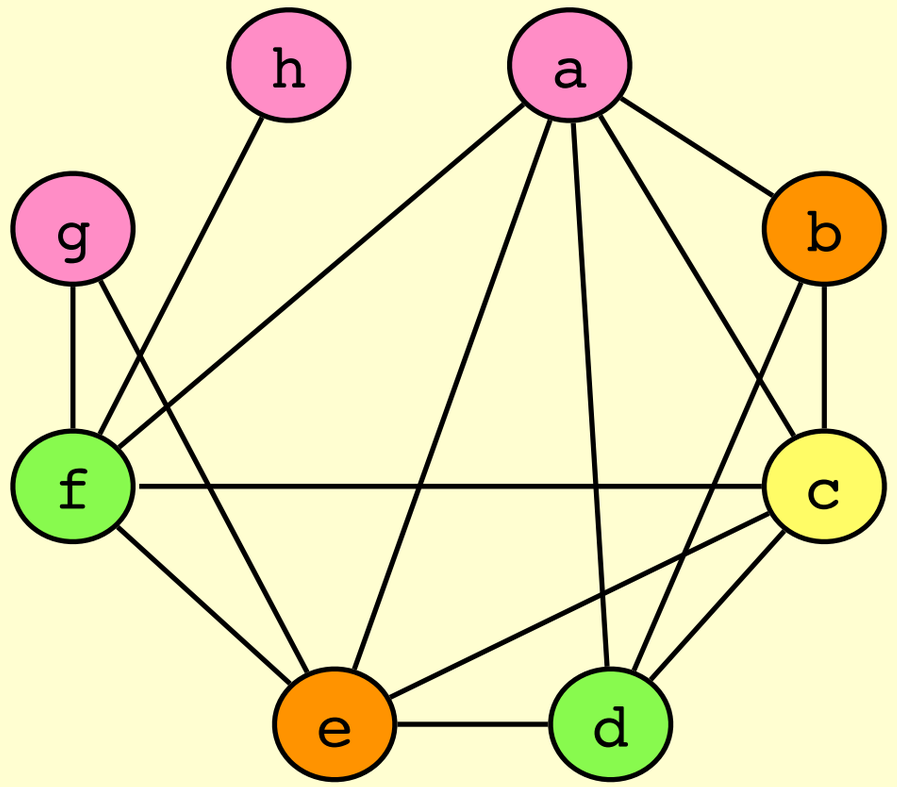
Register allocation

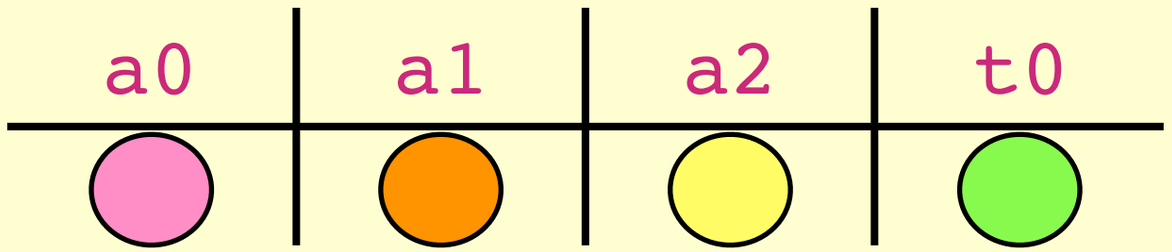
```
int main (int a, int b, int c) {
                        //a b c
  int d = a + b; //a b c d
  int e = d + b; //a c d e
  int f = d + a; //a c e f
  int g = a + c; //e f g
  int h = e + g; //f h
  return f + h;
}
```



Register allocation

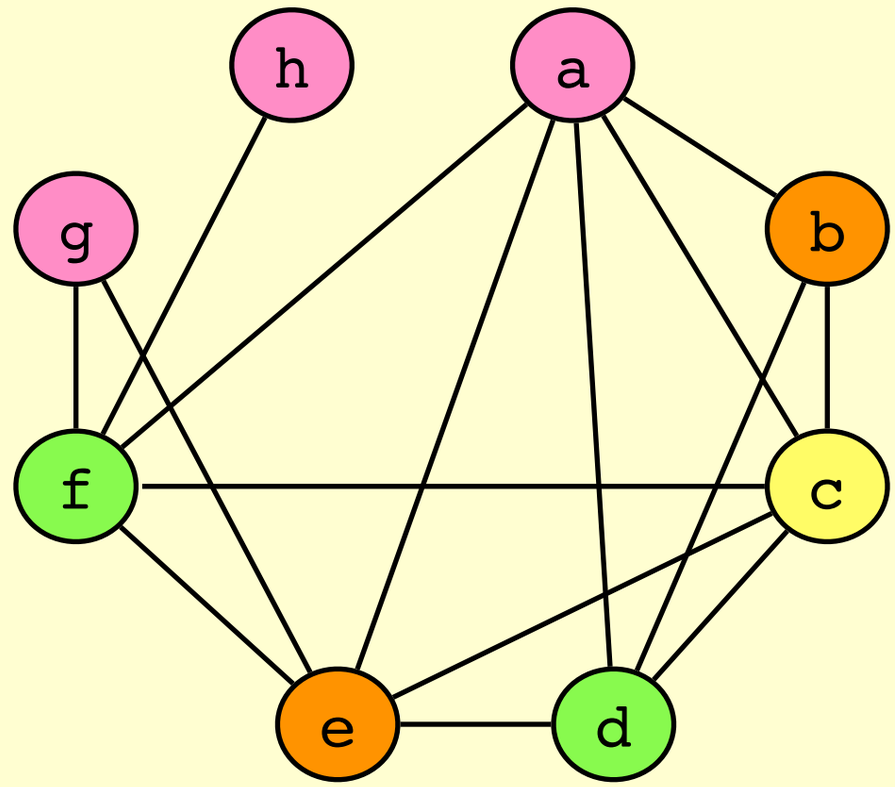
```
int main (int a, int b, int c) {
                        //a b c
   int d = a + b; //a b c d
   int e = d + b; //a c d e
   int f = d + a; //a c e f
   int g = a + c; //e f g
   int h = e + g; //f h
   return f + h;
}
```

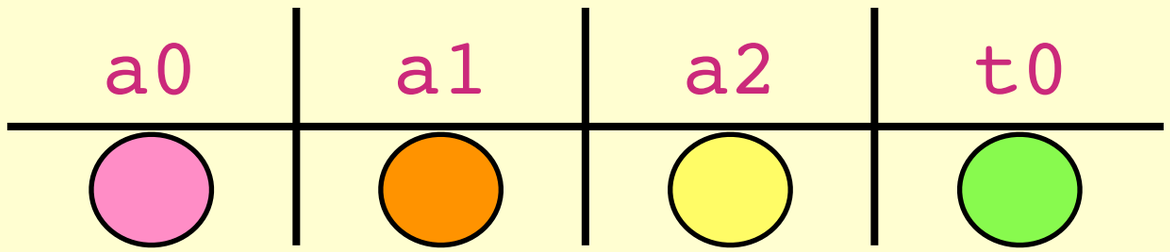




Register allocation

```
int main (a0, a1, a2) {

   ADD t0, a0, a1
   ADD a1, t0, a1
   ADD t0, t0, a0
   ADD a0, a0, a2
   ADD a0, a1, a0
   ADD a0, t0, a0
}
```





Exercise!

Task: Work out which variables are live at each program point. Then deduce the register interference graph.

```
int main (int a) {
  int b, c, d, e, f, g, h;

  b = a + 1;
  c = b * a;
  d = a - 2;
  e = b * c;
  f = c + 1;
  g = d * f;
  h = e + g;
  a = f - 2;
  b = g + 1;
  c = h + a;
  return c - b;
}
```

Exercise!

Task: Work out which variables are live at each program point. Then deduce the register interference graph.

```
int main (int a) {
  int b, c, d, e, f, g, h;
                        //a
  b = a + 1;
  c = b * a;
  d = a - 2;
  e = b * c;
  f = c + 1;
  g = d * f;
  h = e + g;
  a = f - 2;
  b = g + 1;
  c = h + a;
  return c - b;
}
```

Exercise!

Task: Work out which variables are live at each program point. Then deduce the register interference graph.

```
int main (int a) {
  int b, c, d, e, f, g, h;
                        //a
  b = a + 1;            //a b
  c = b * a;
  d = a - 2;
  e = b * c;
  f = c + 1;
  g = d * f;
  h = e + g;
  a = f - 2;
  b = g + 1;
  c = h + a;
  return c - b;
}
```

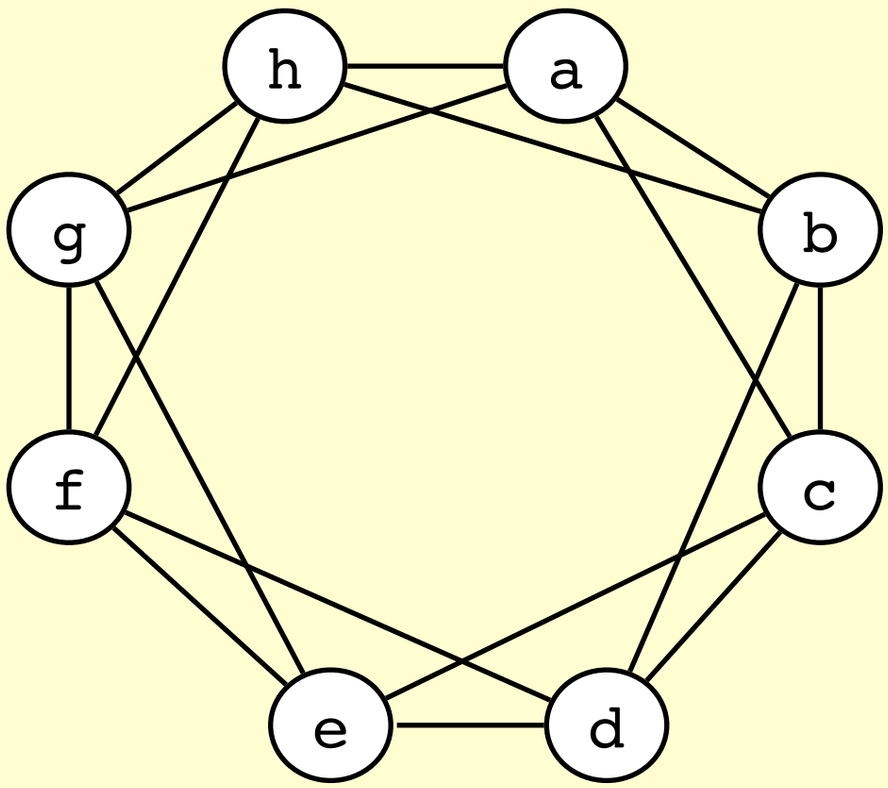
Exercise!

Task: Work out which variables are live at each program point. Then deduce the register interference graph.

```
int main (int a) {
  int b, c, d, e, f, g, h;
                        //a
  b = a + 1;            //a b
  c = b * a;            //a b c
  d = a - 2;
  e = b * c;
  f = c + 1;
  g = d * f;
  h = e + g;
  a = f - 2;
  b = g + 1;
  c = h + a;
  return c - b;
}
```

Exercise!

Task: Work out which variables are live at each program point. Then deduce the register interference graph.

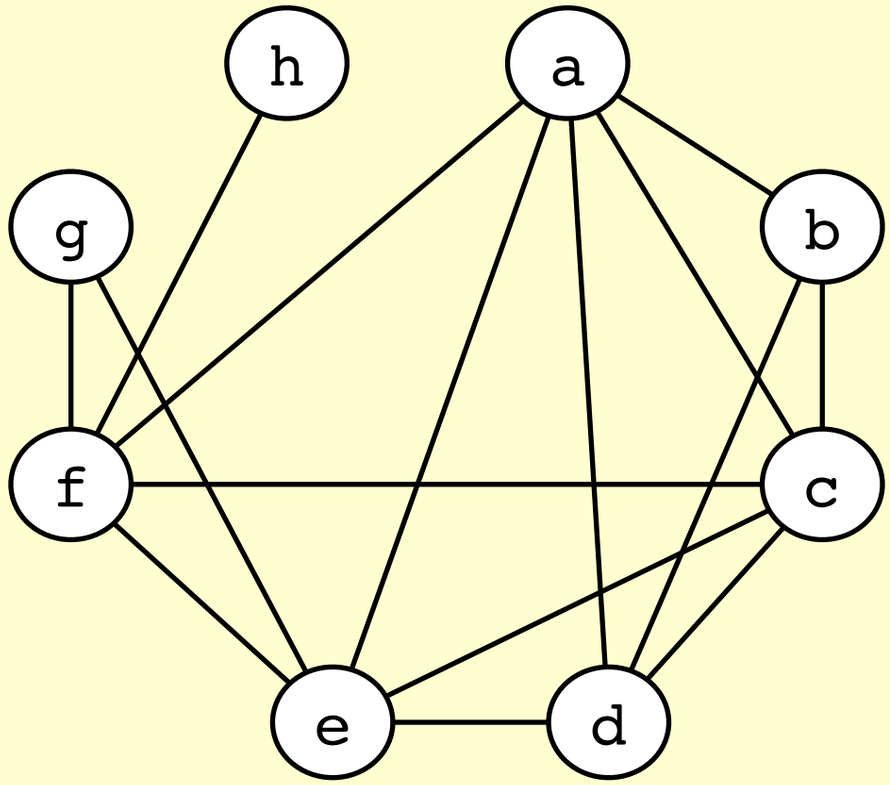



```
int main (int a) {
  int b, c, d, e, f, g, h;
                        //a
  b = a + 1;            //a b
  c = b * a;            //a b c
  d = a - 2;            //b c d
  e = b * c;            //c d e
  f = c + 1;            //d e f
  g = d * f;            //e f g
  h = e + g;            //f g h
  a = f - 2;            //g h a
  b = g + 1;            //h a b
  c = h + a;            //c b
  return c - b;
}
```

Heuristic colouring

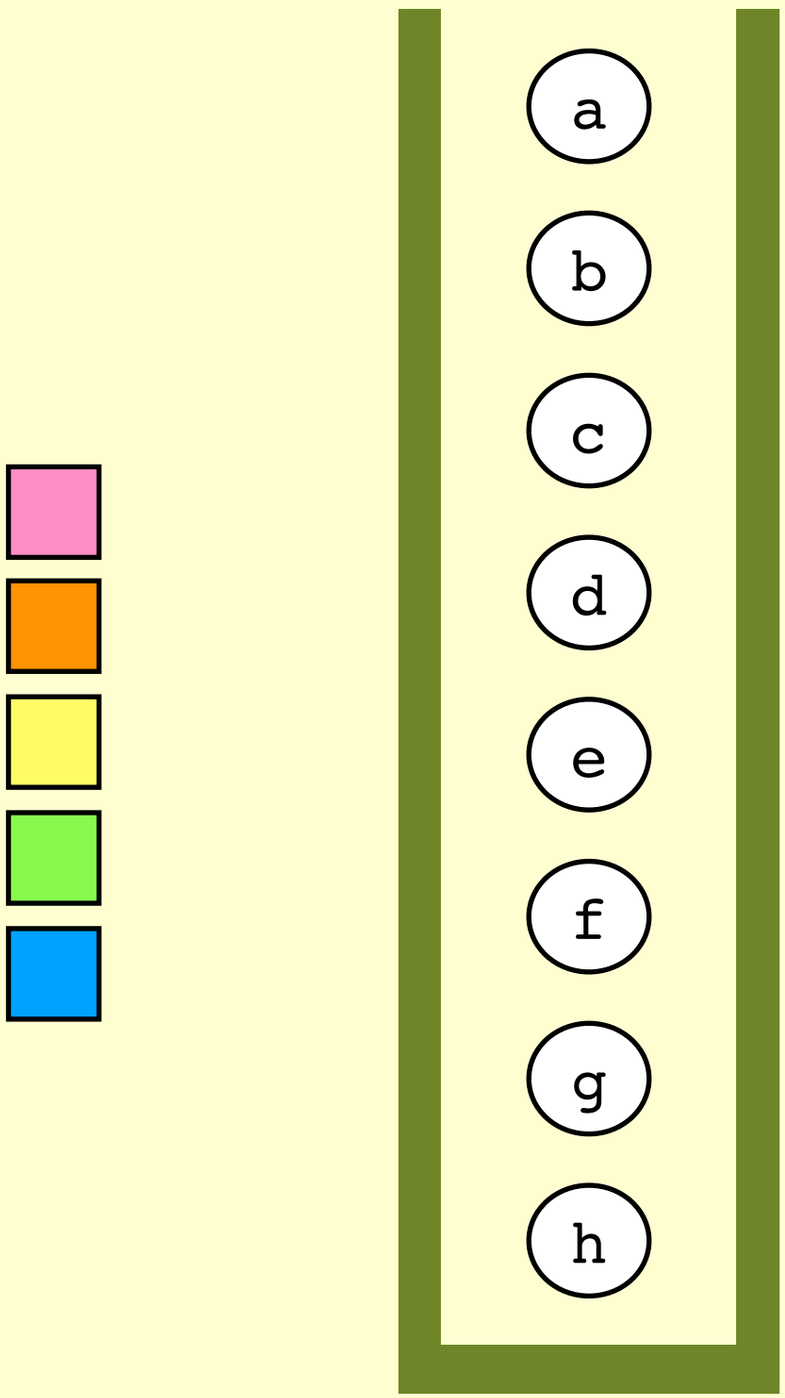
- We know that graph-colouring is hard.
- So let's use a simple scheme that does a pretty good job (but might use more colours than strictly necessary).

Heuristic colouring



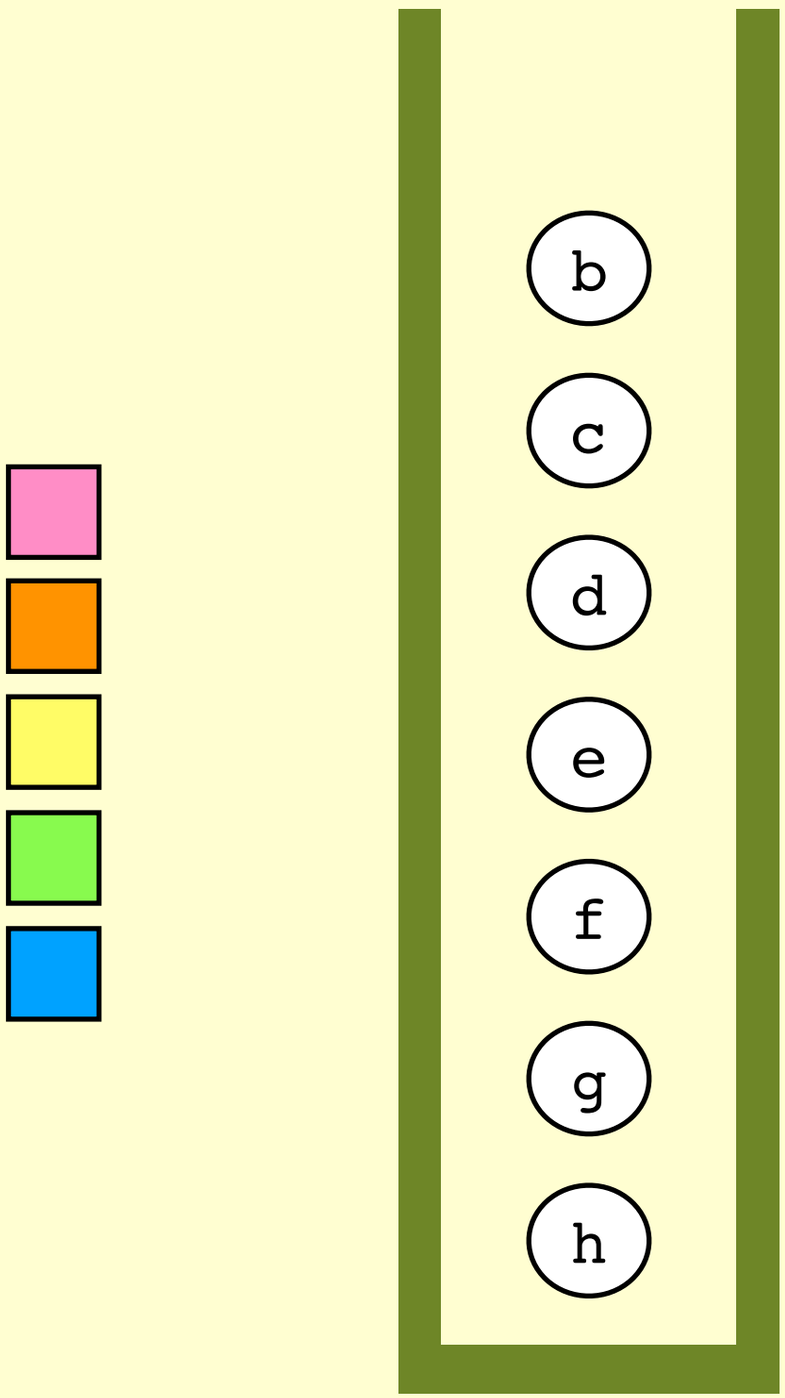
1. Remove node with fewest neighbours.
2. Repeat

Heuristic colouring



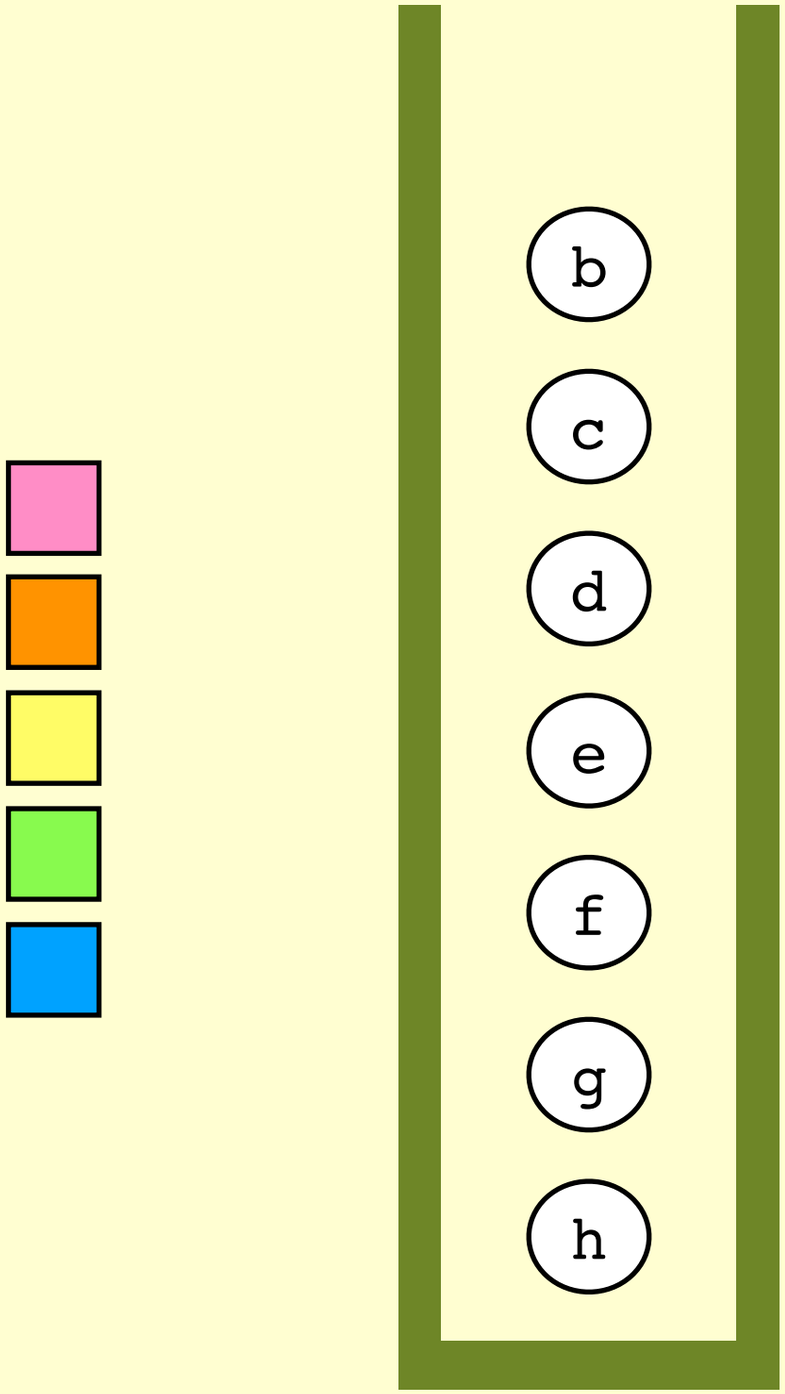
1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

Heuristic colouring



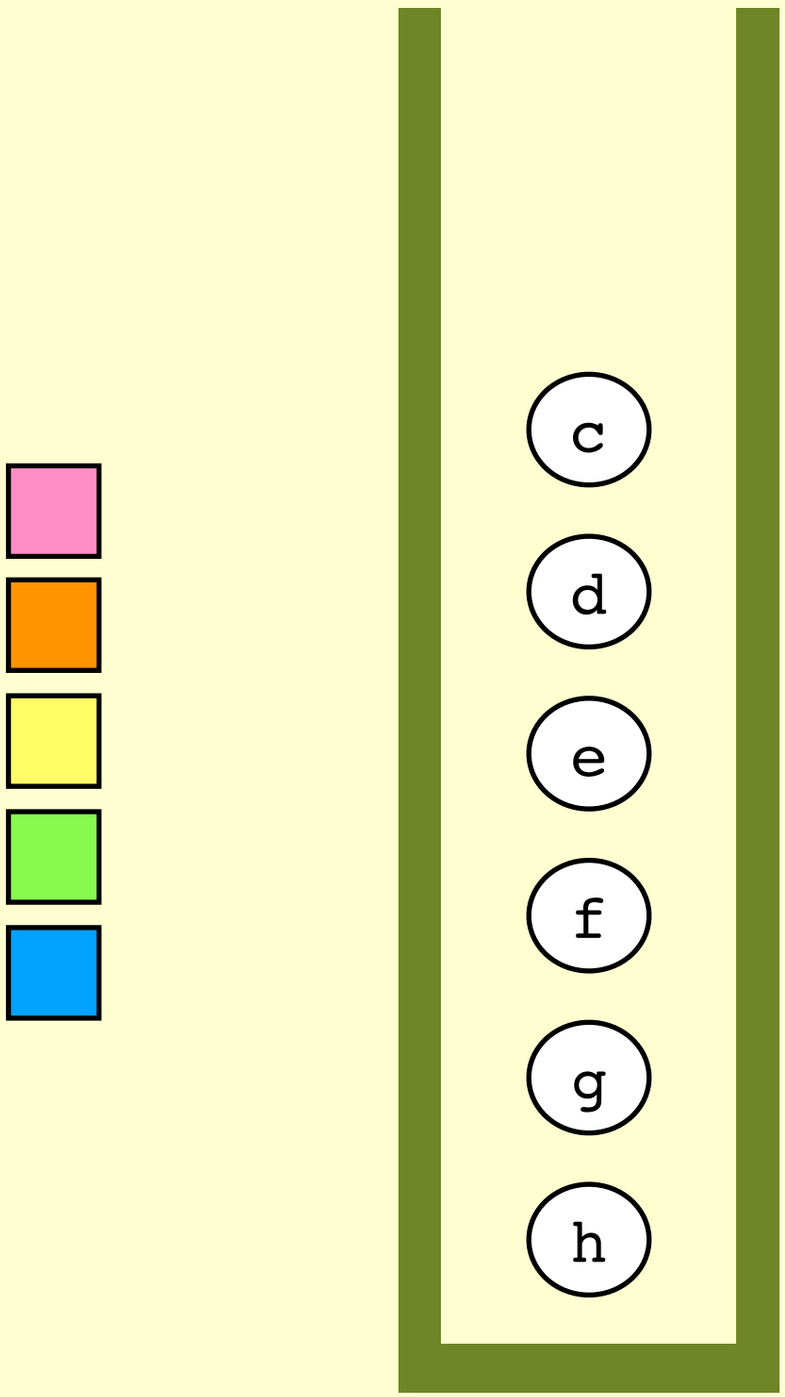
1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

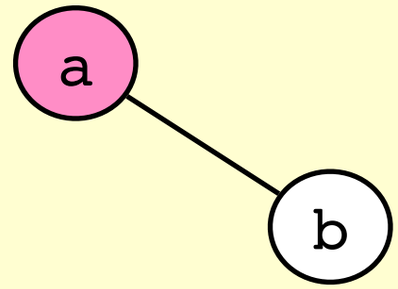
Heuristic colouring



1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

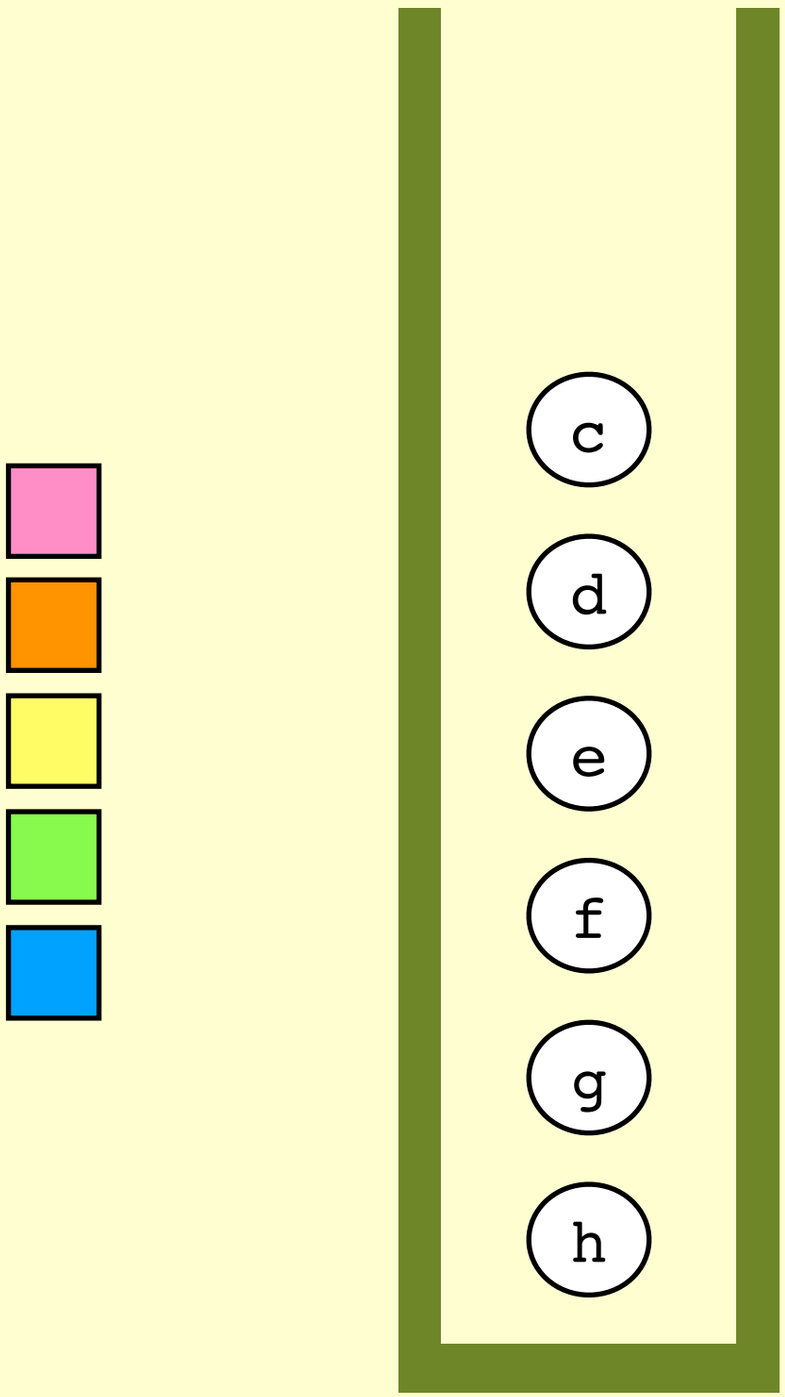
Heuristic colouring

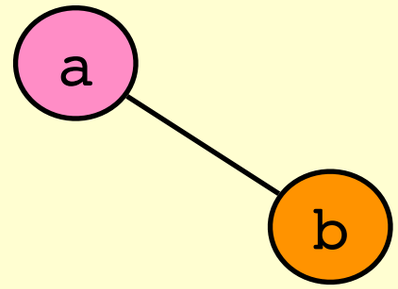




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

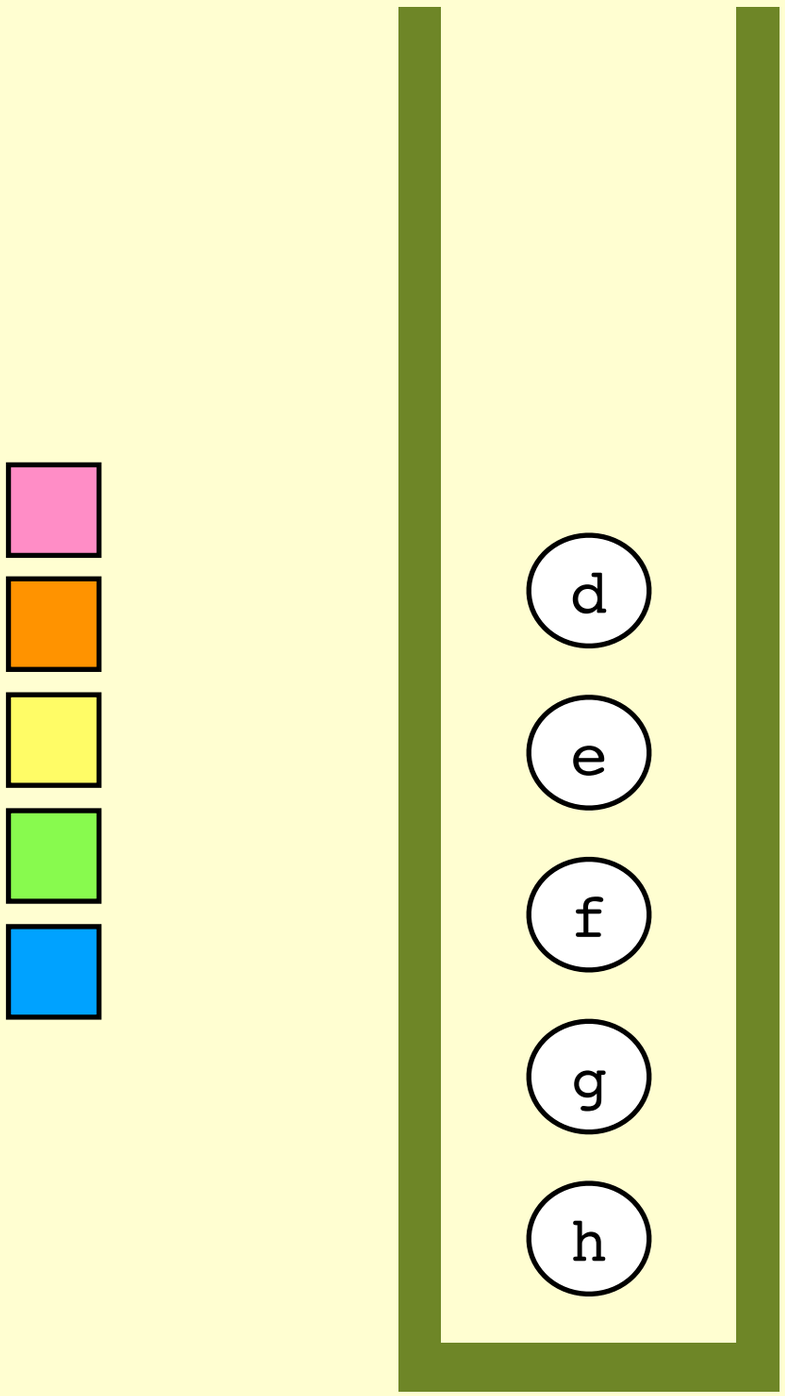
Heuristic colouring

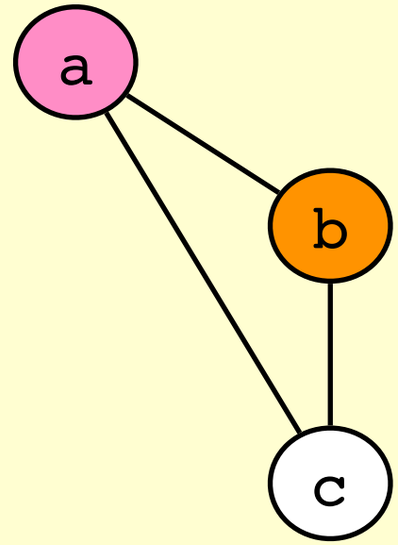




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

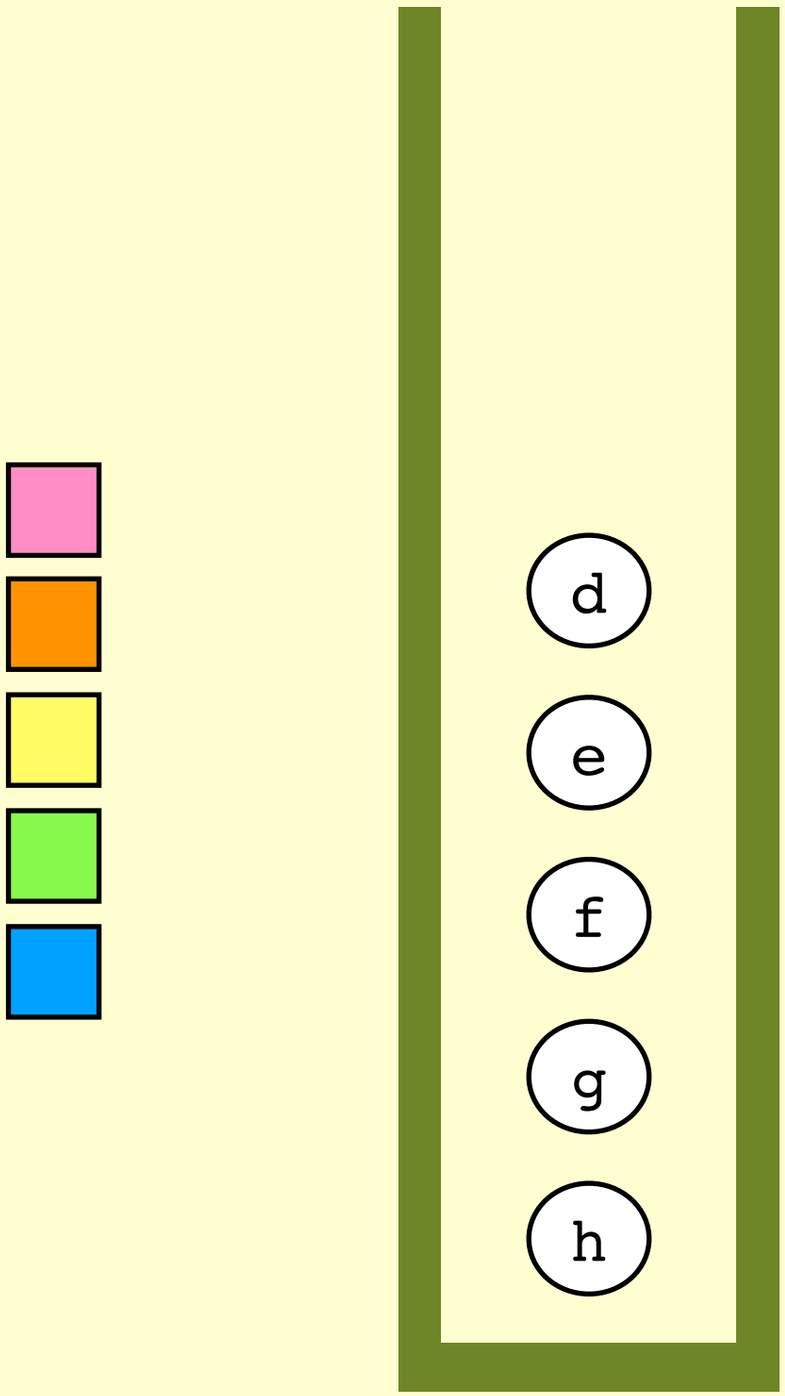
Heuristic colouring

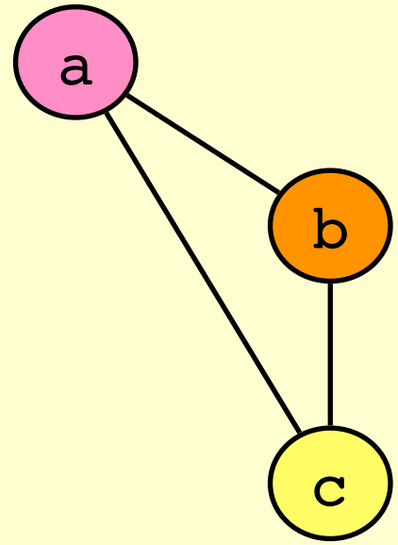




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

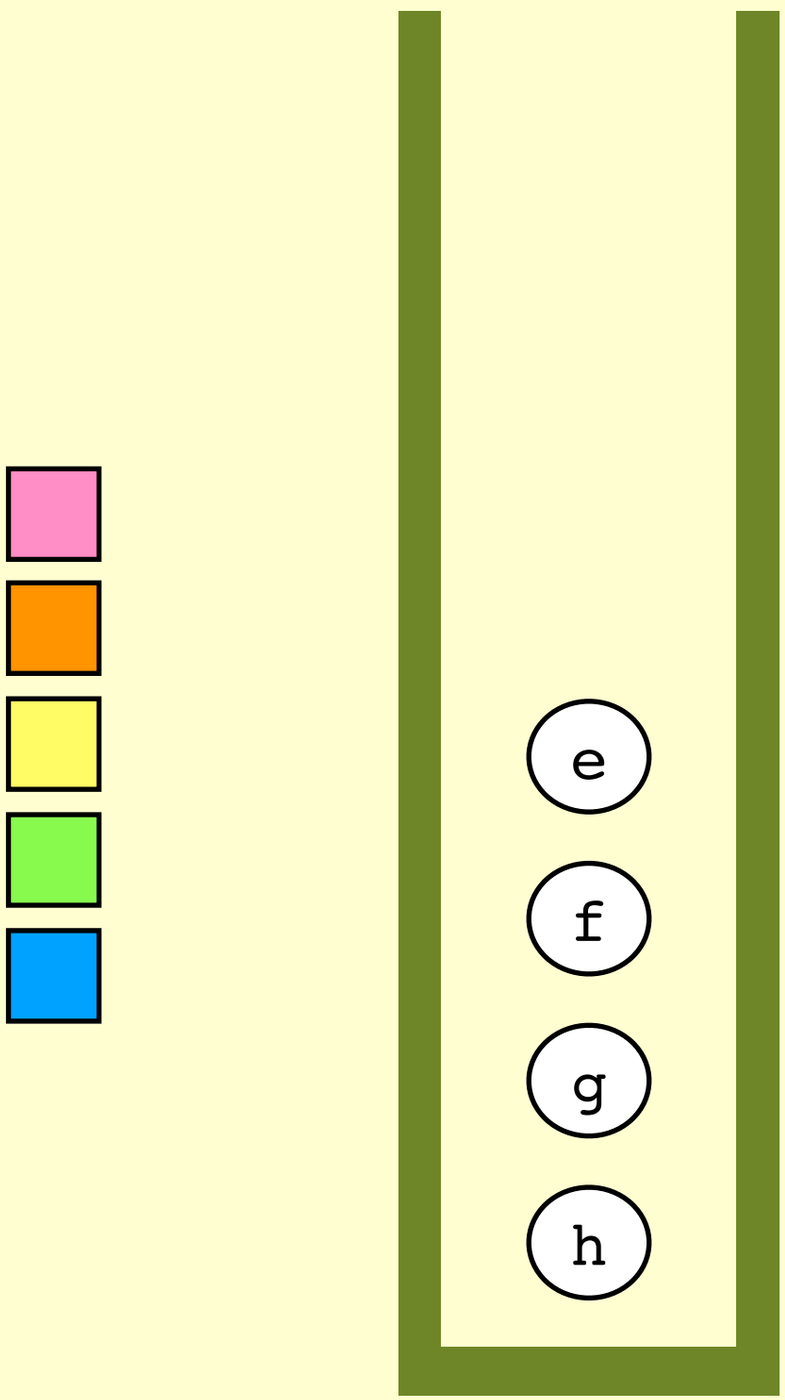
Heuristic colouring

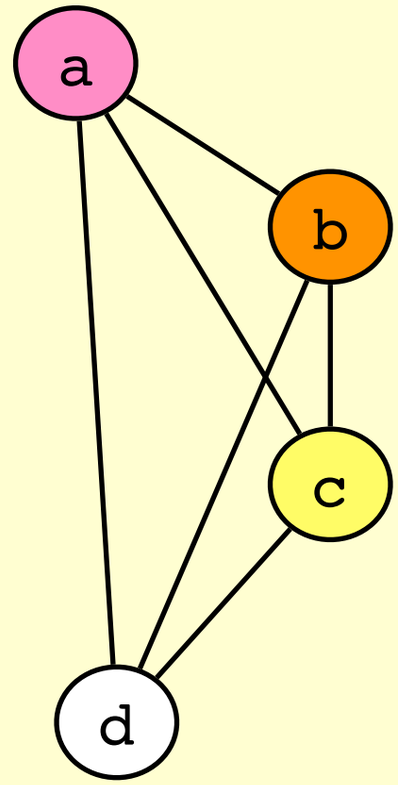




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

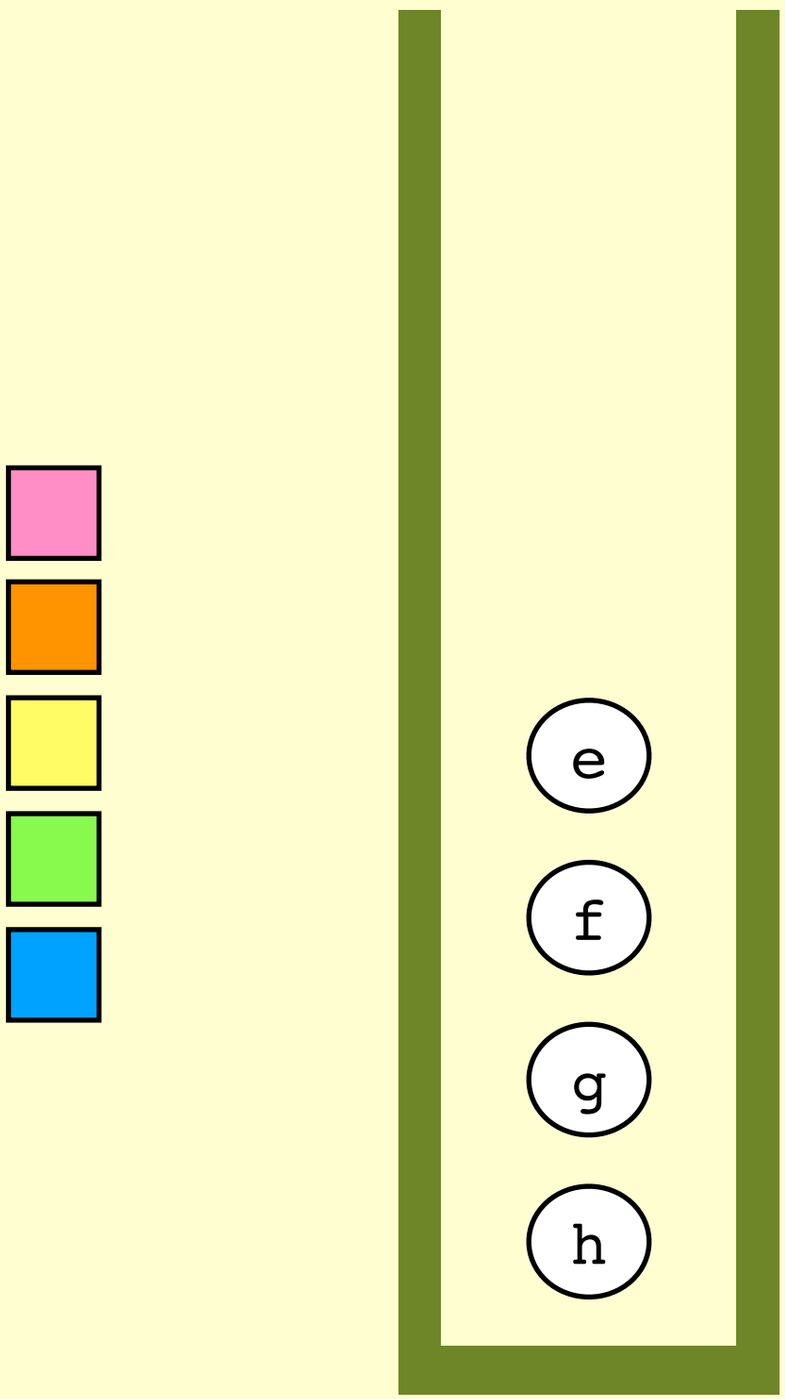
Heuristic colouring

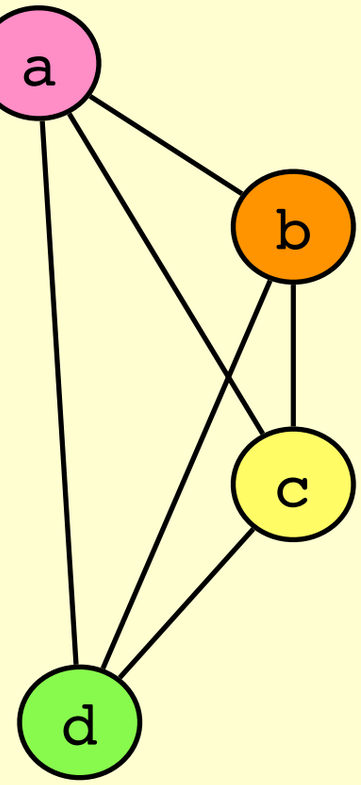




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

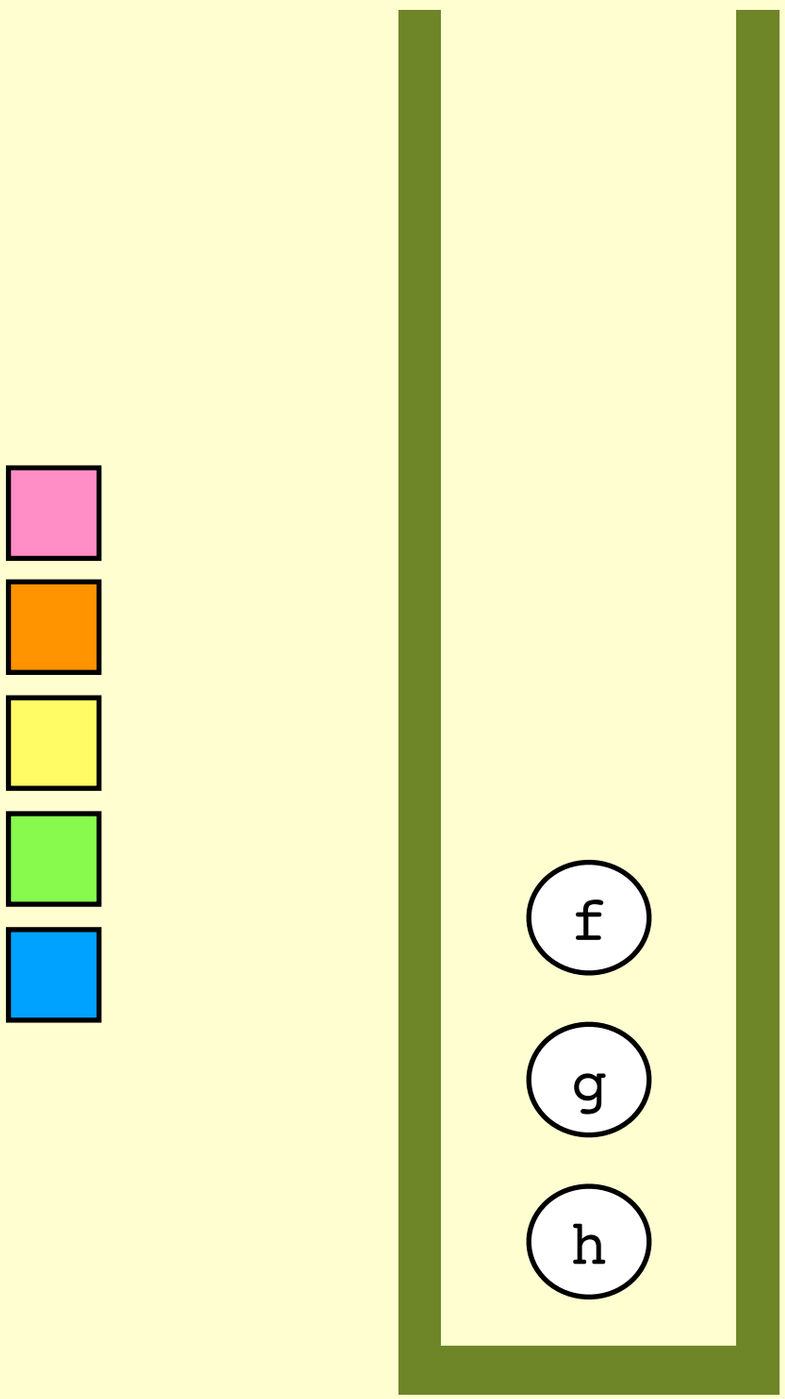
Heuristic colouring

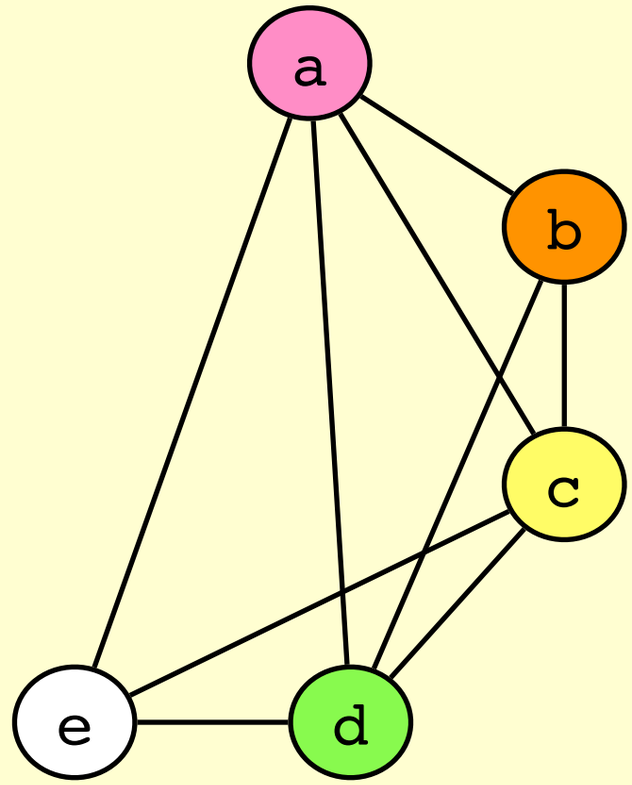




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

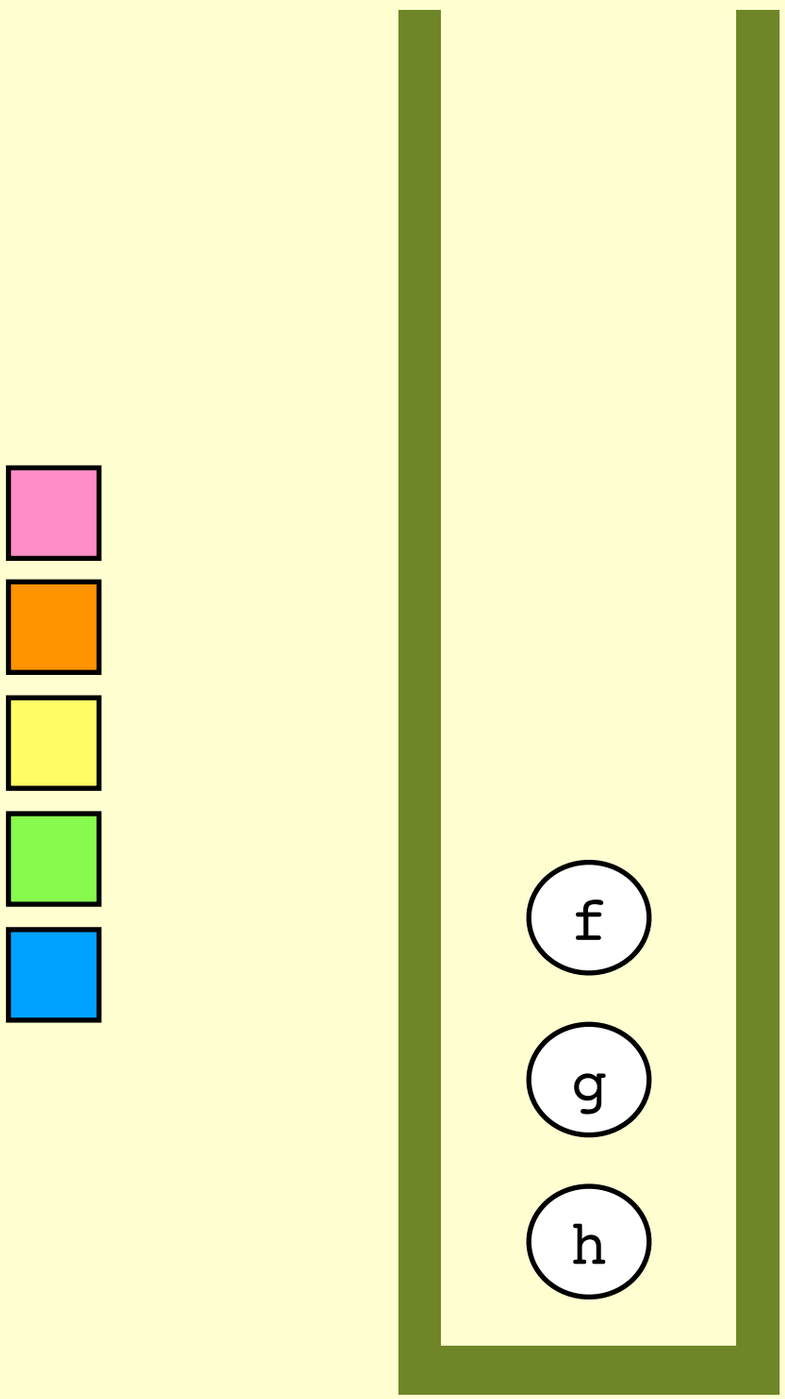
Heuristic colouring

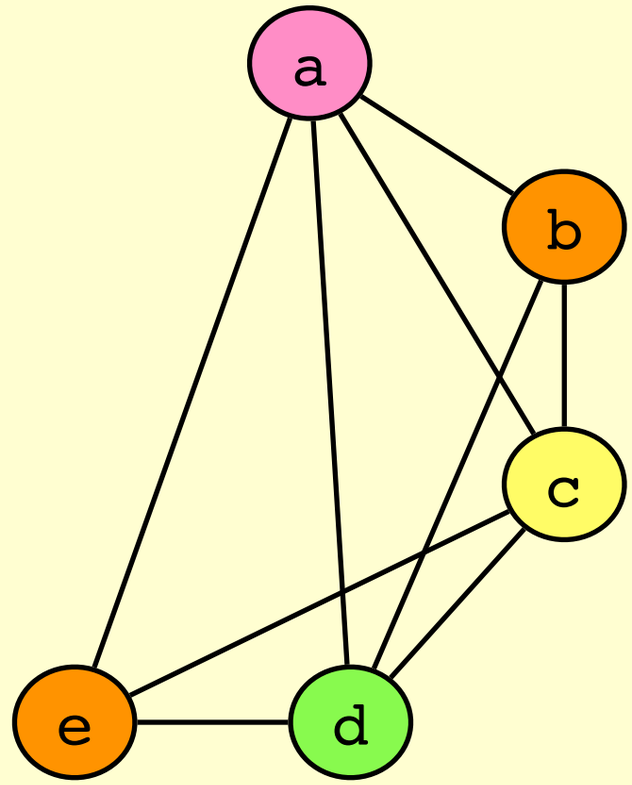




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

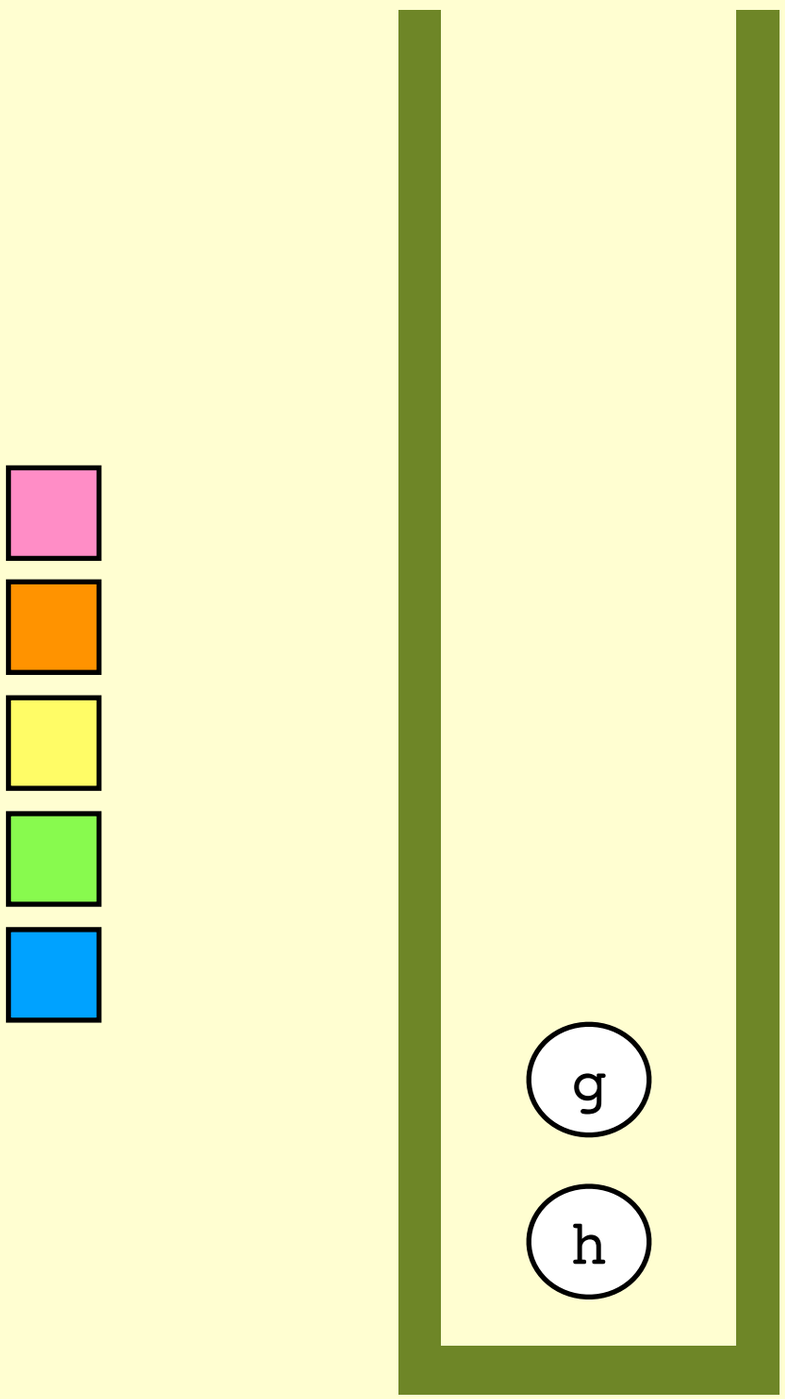
Heuristic colouring

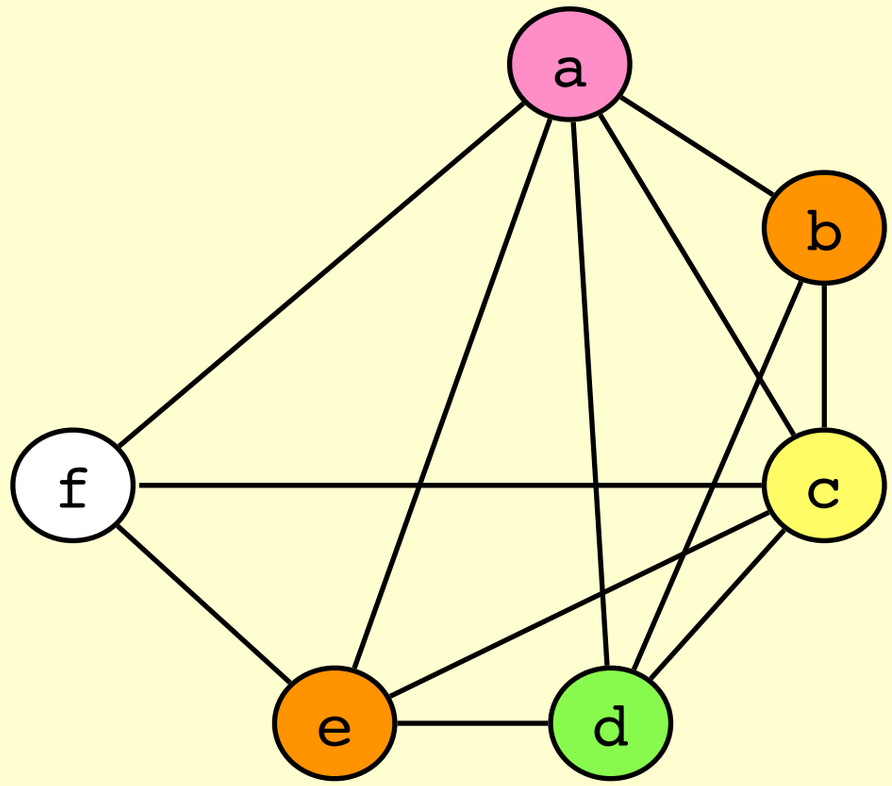




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

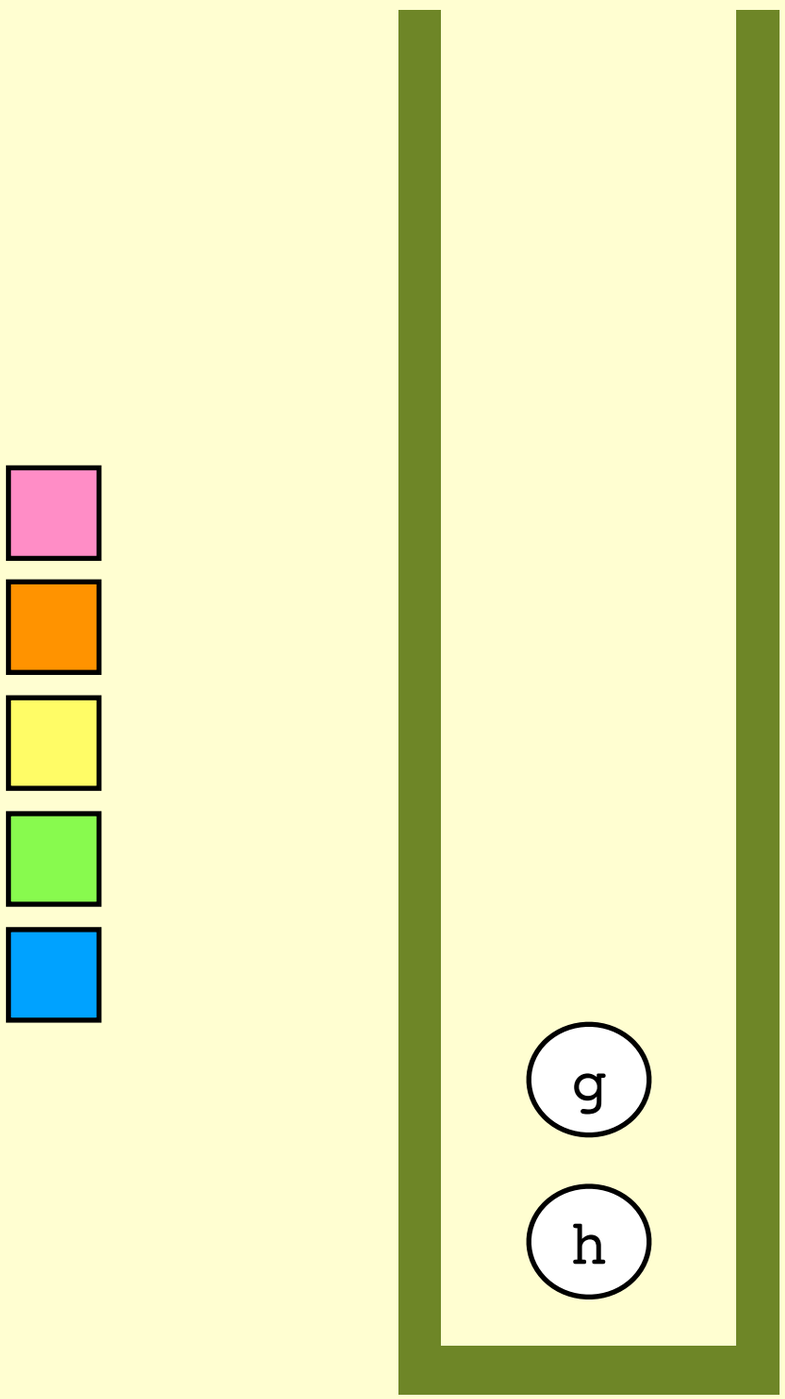
Heuristic colouring

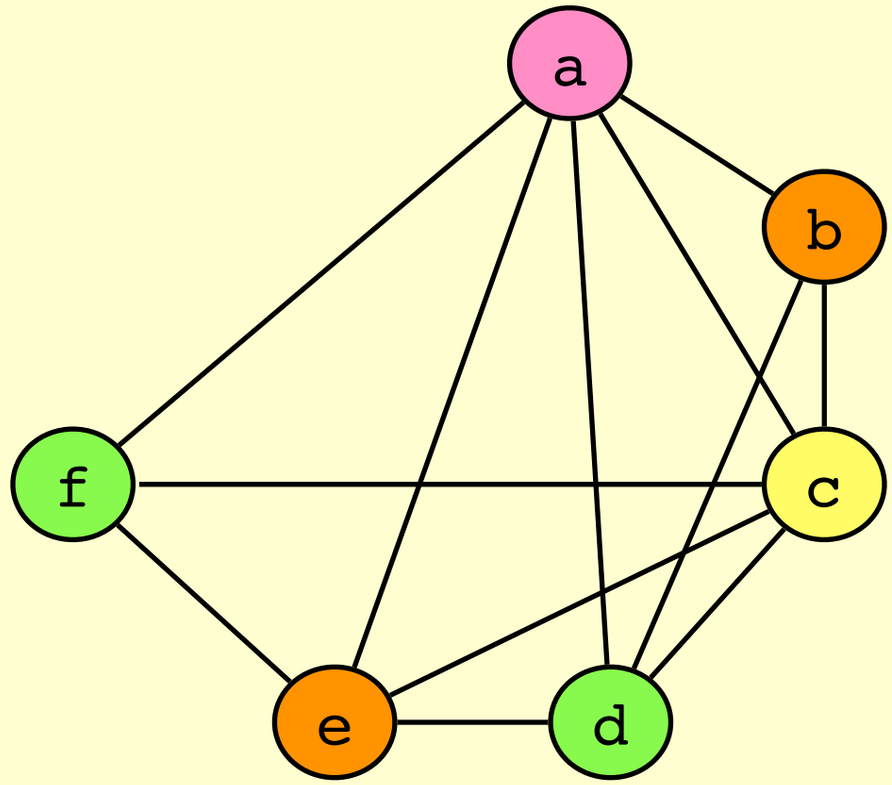




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

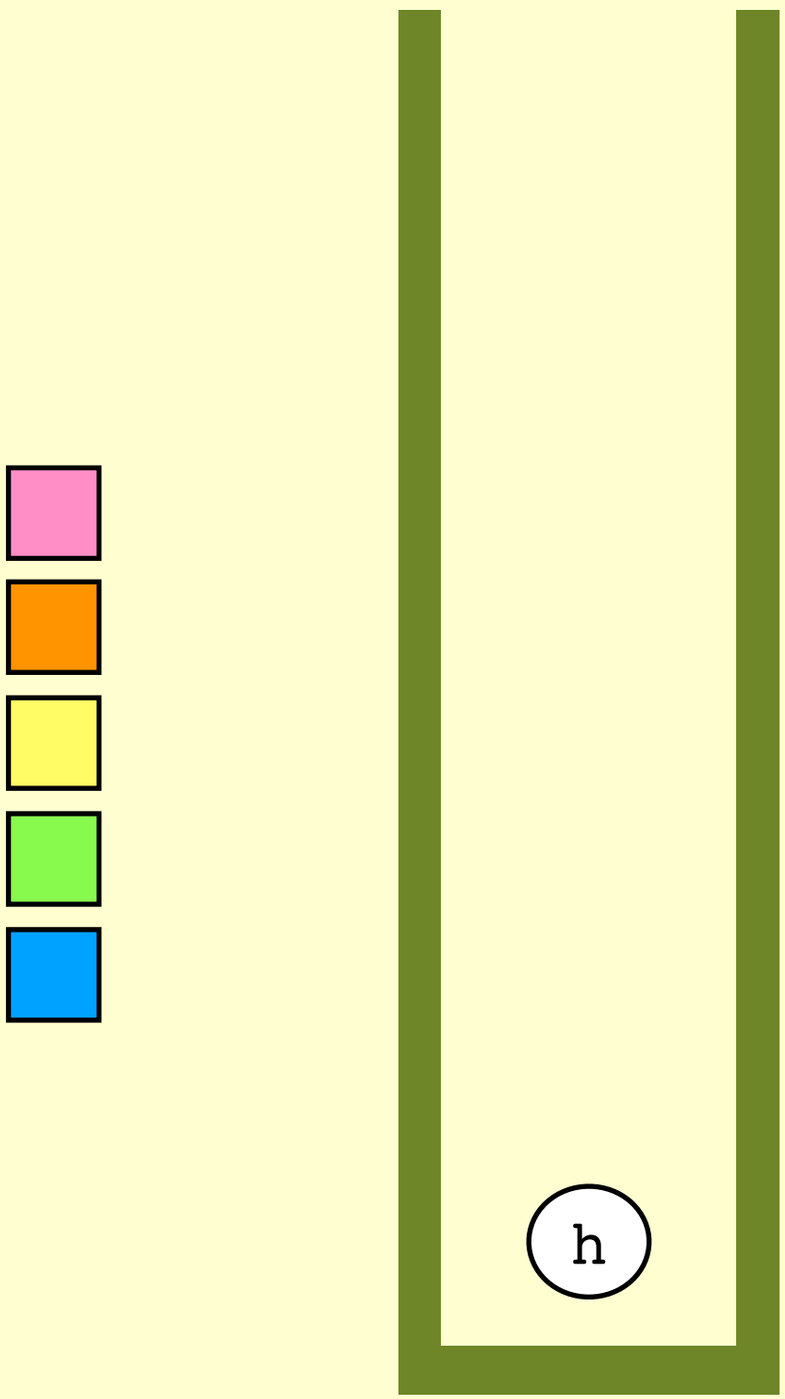
Heuristic colouring

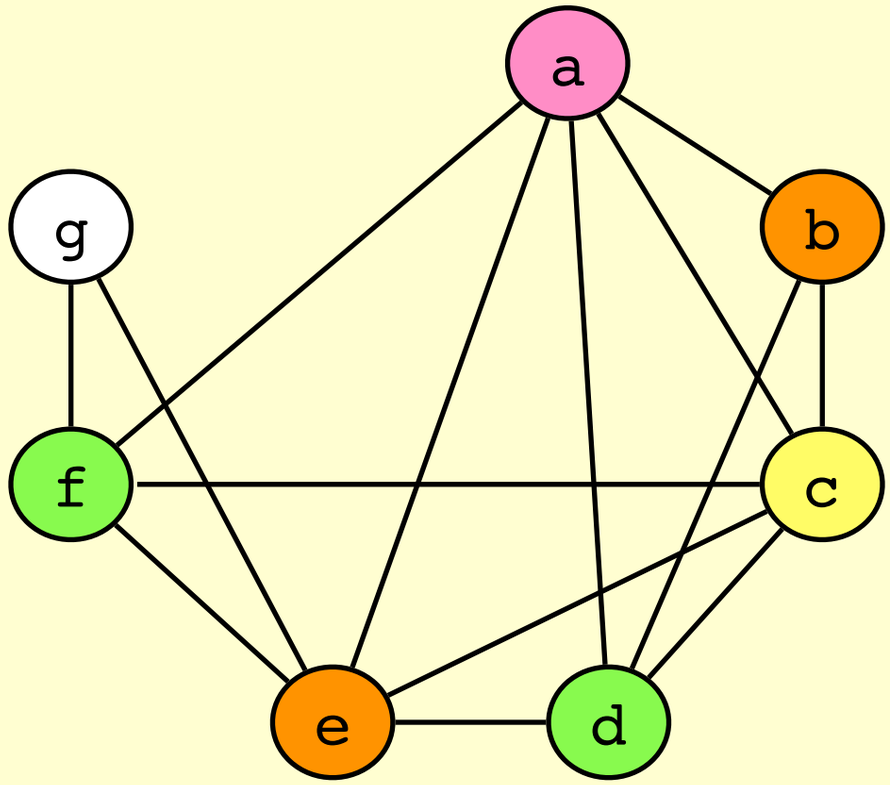




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

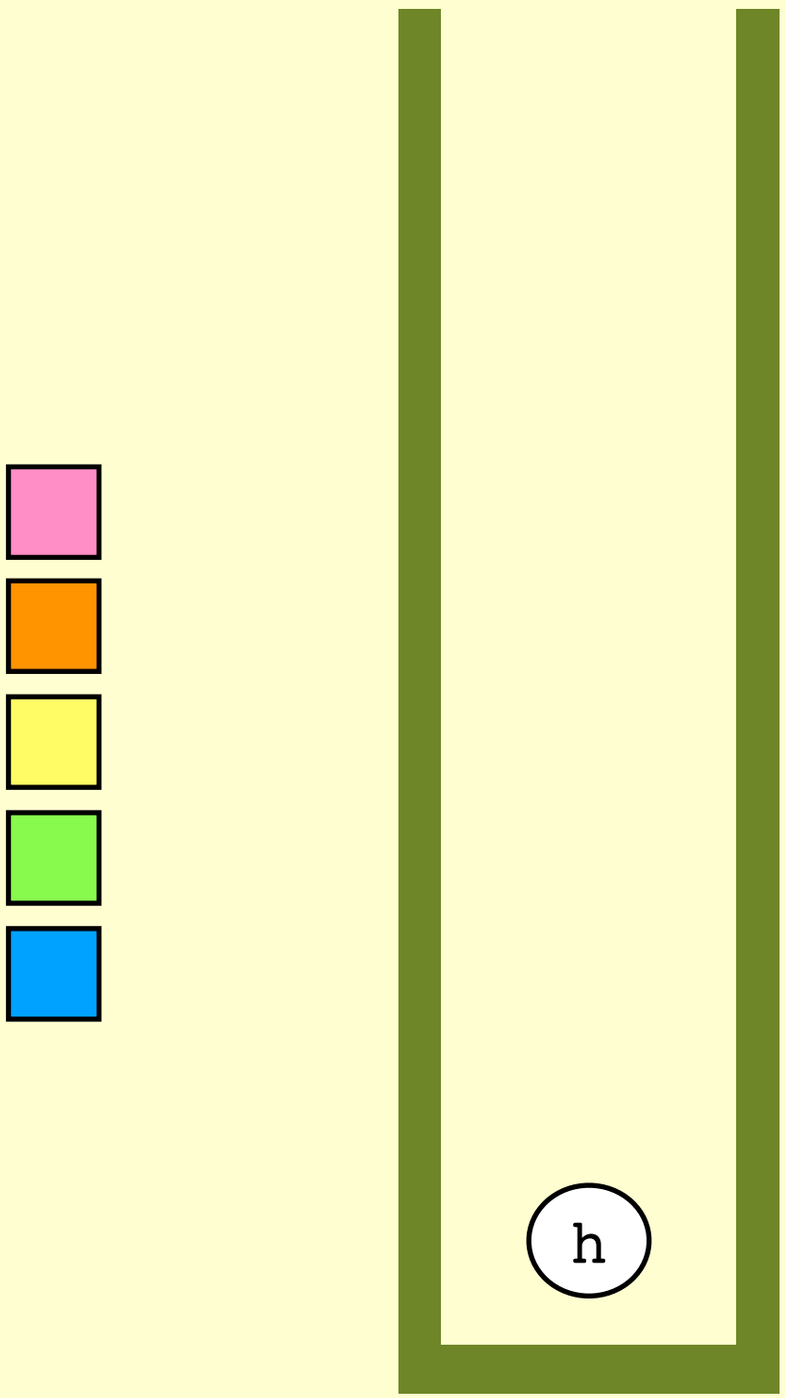
Heuristic colouring

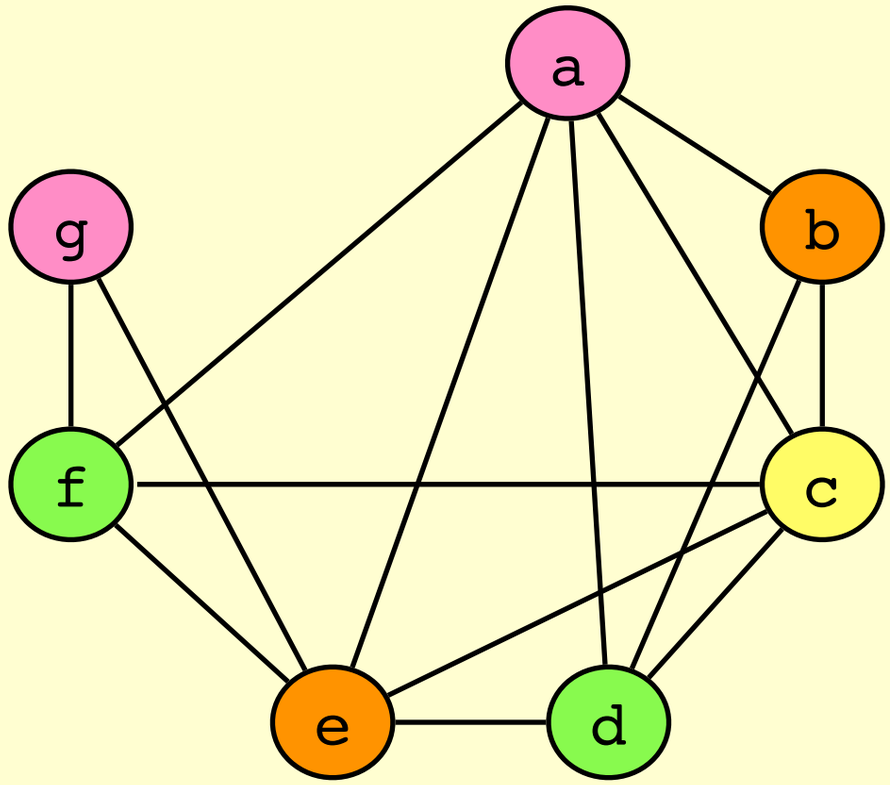




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

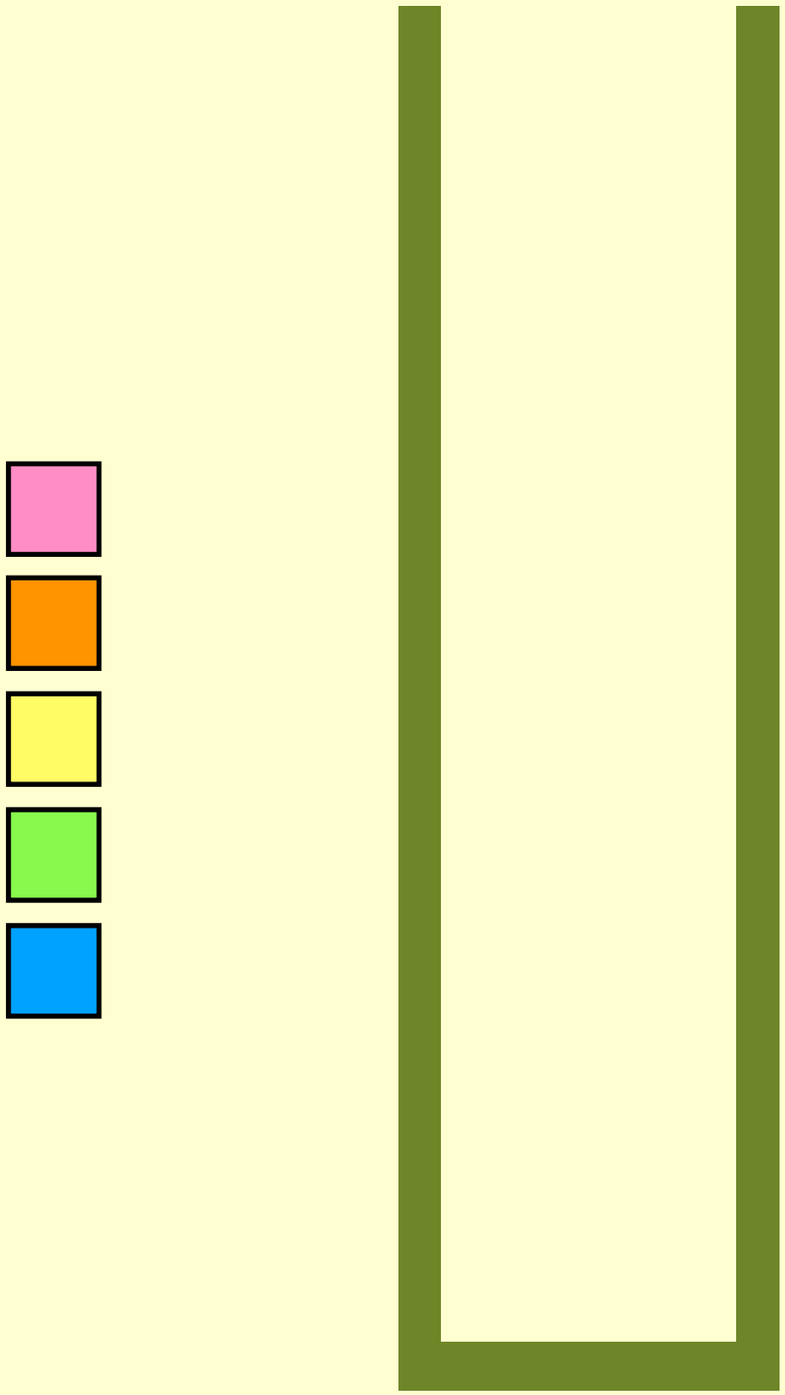
Heuristic colouring

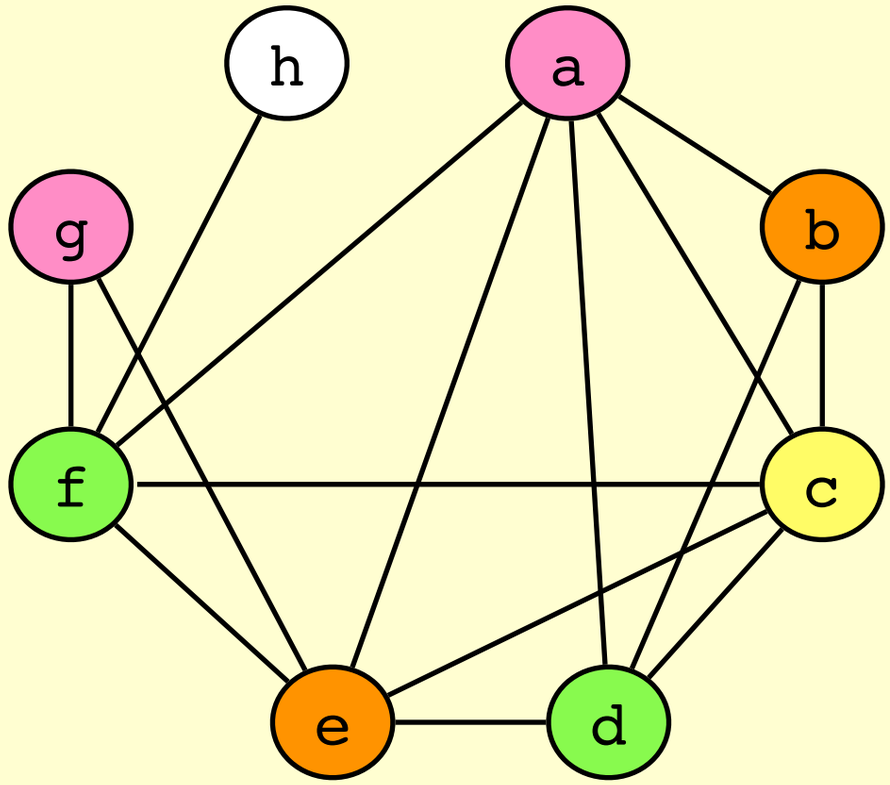




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

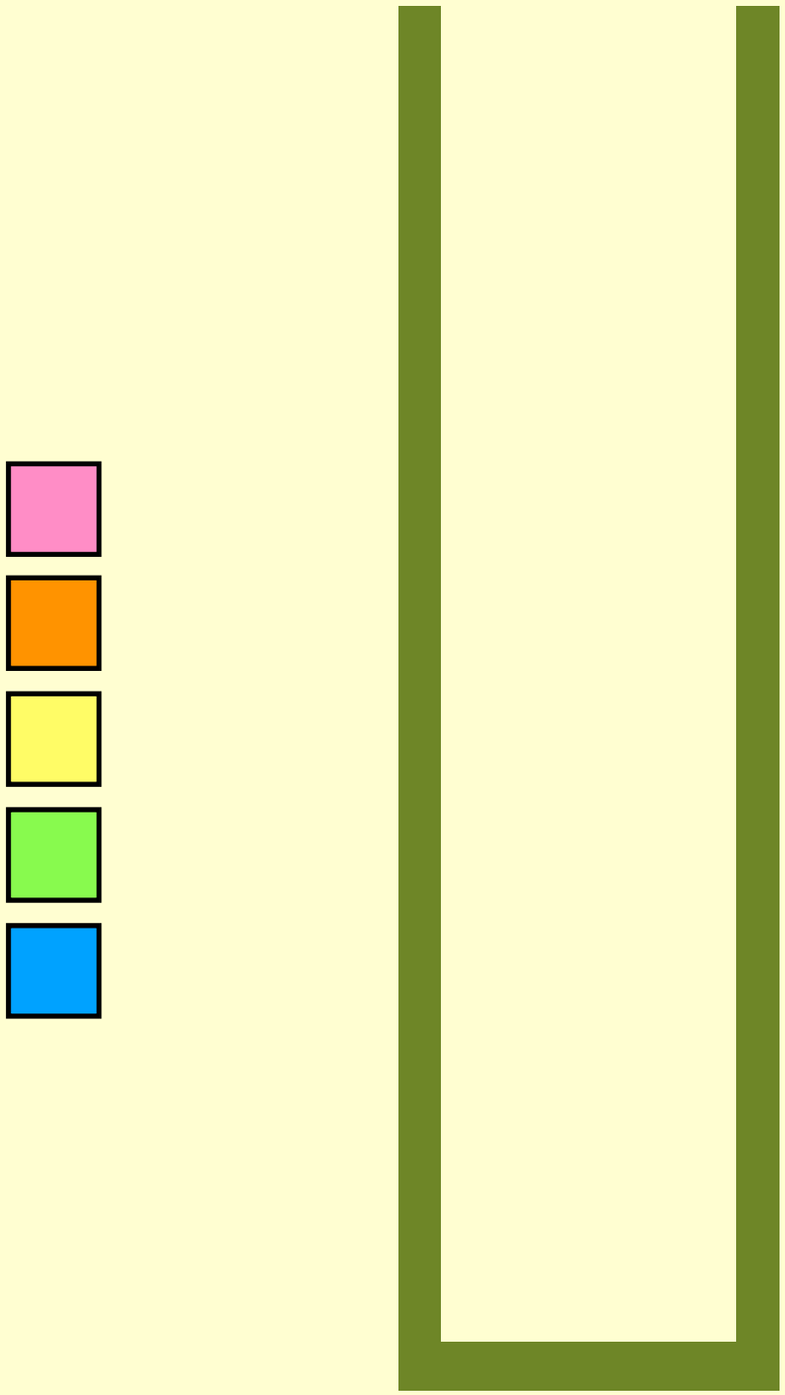
Heuristic colouring

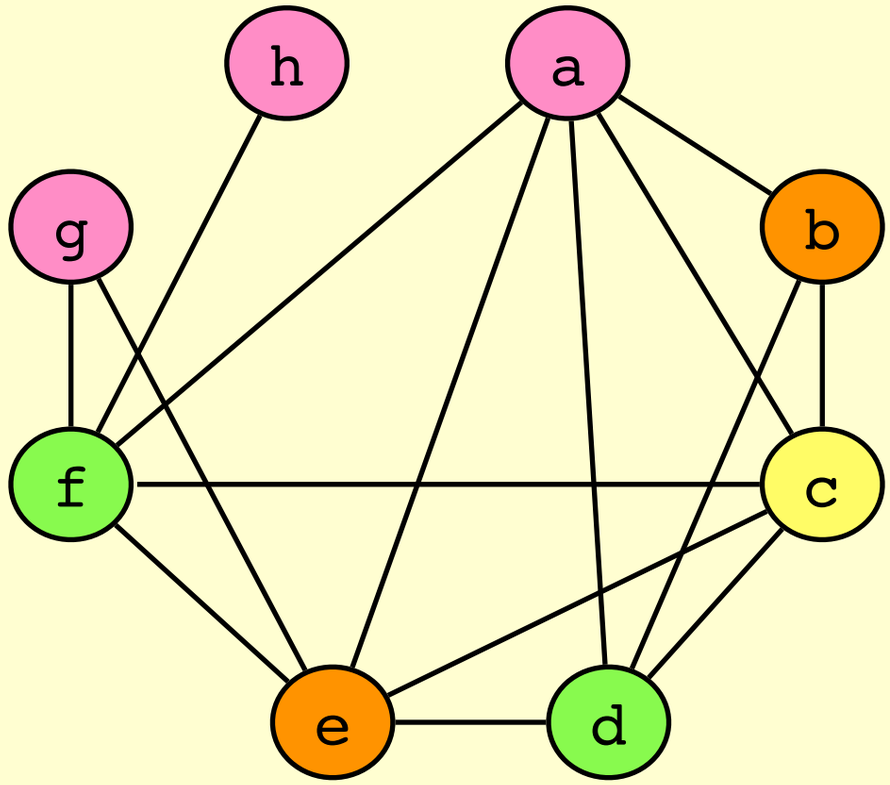




1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

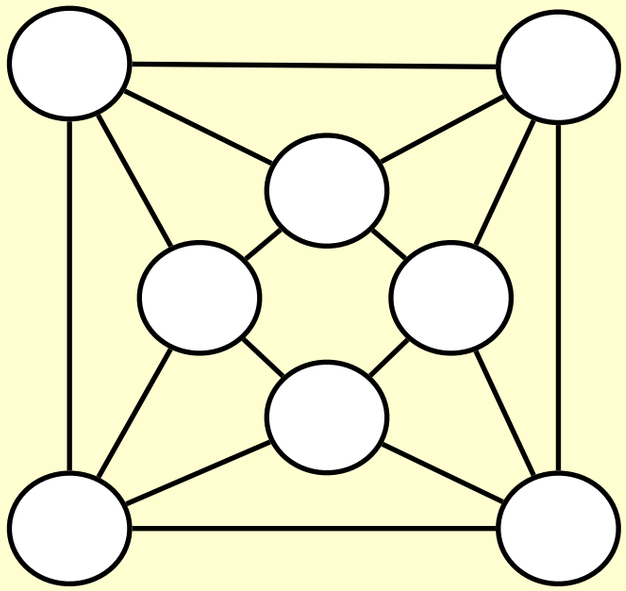
Heuristic colouring



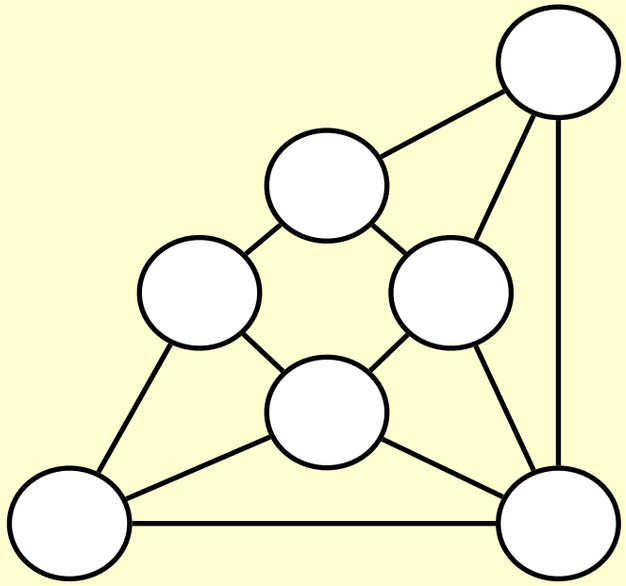


1. Remove node with fewest neighbours.
2. Repeat until graph is empty.
3. Colour nodes while rebuilding graph, always using first feasible colour.

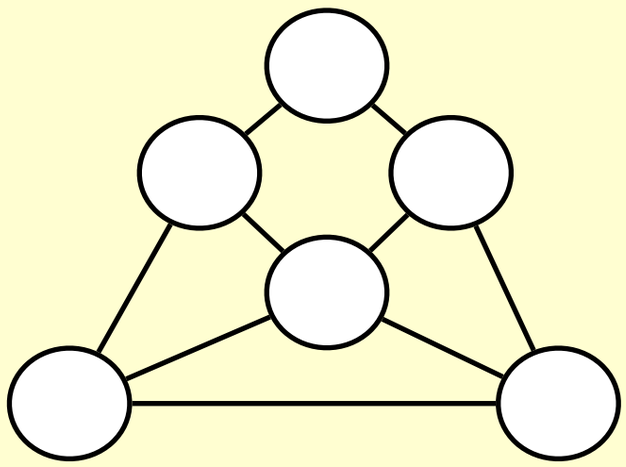
When the heuristic fails



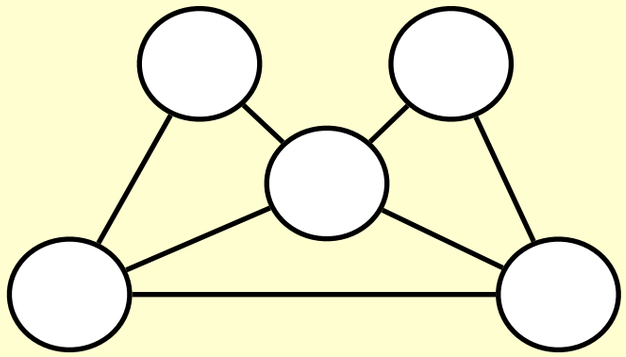
When the heuristic fails



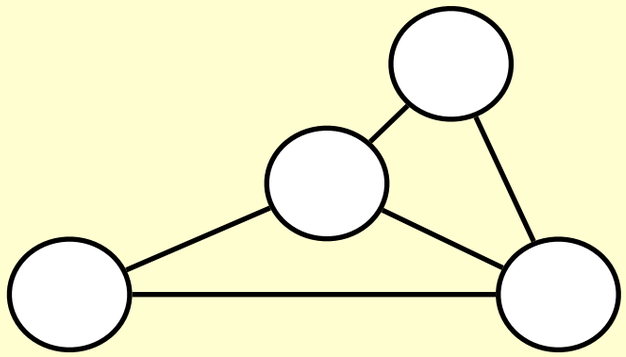
When the heuristic fails



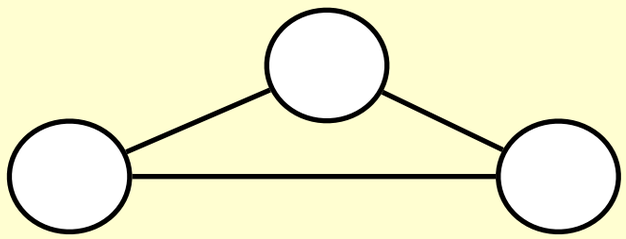
When the heuristic fails



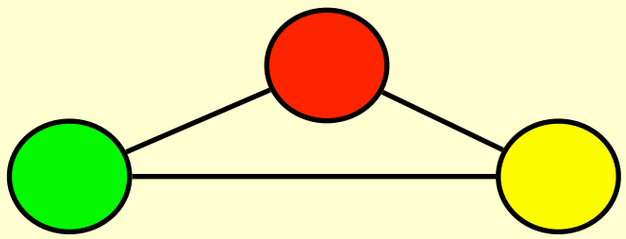
When the heuristic fails



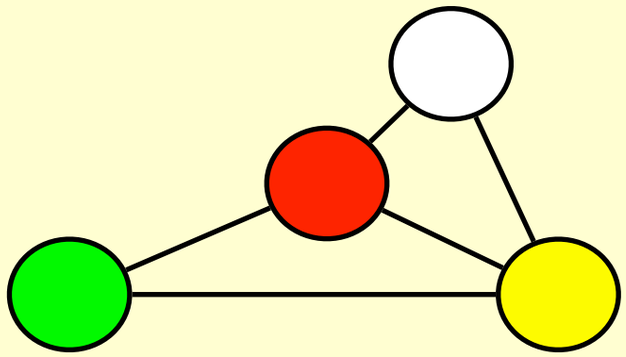
When the heuristic fails



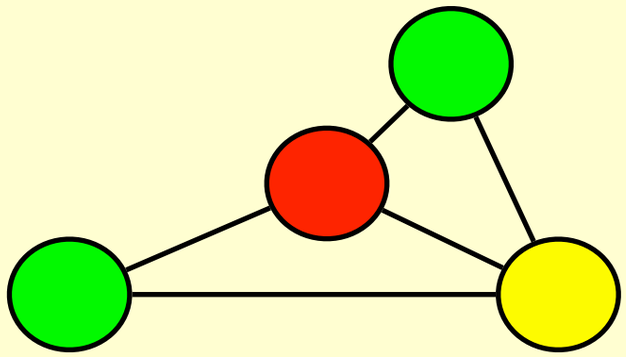
When the heuristic fails



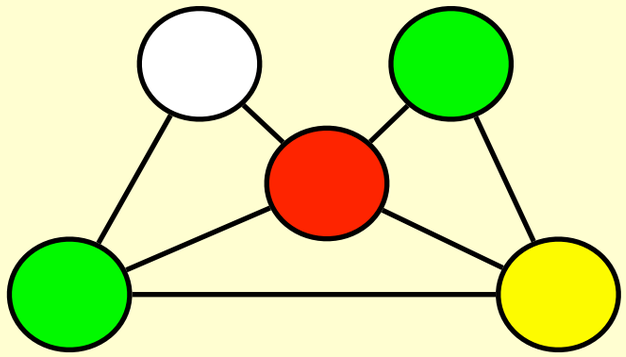
When the heuristic fails



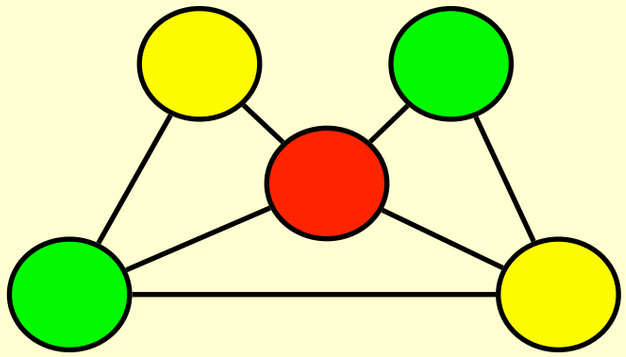
When the heuristic fails



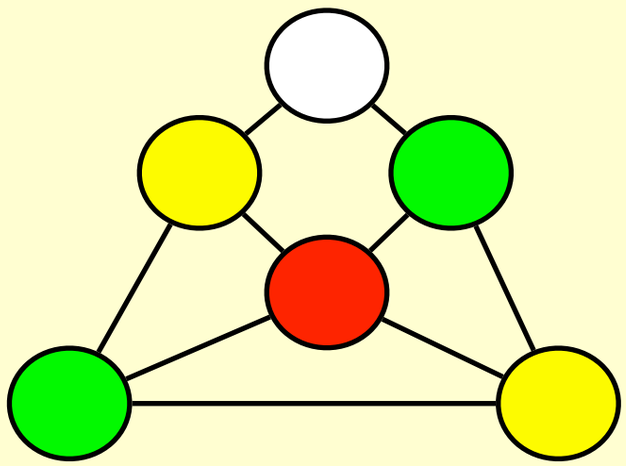
When the heuristic fails



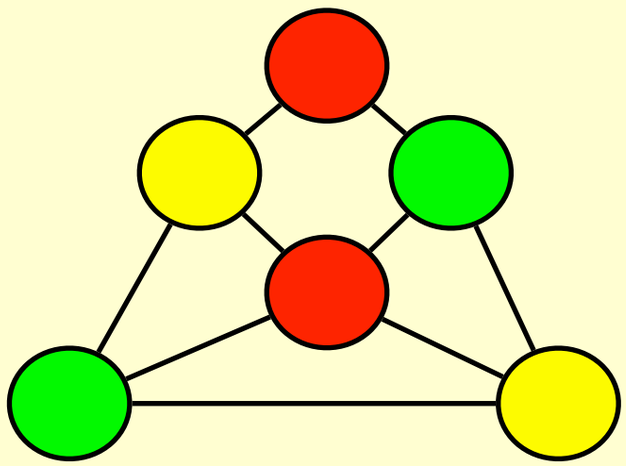
When the heuristic fails



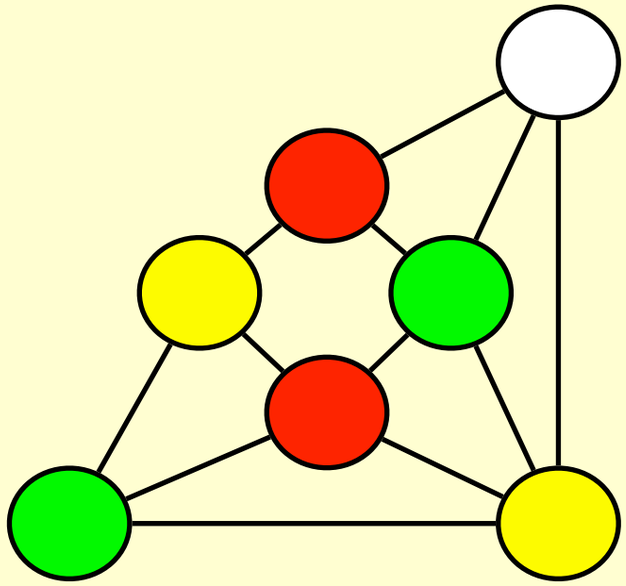
When the heuristic fails



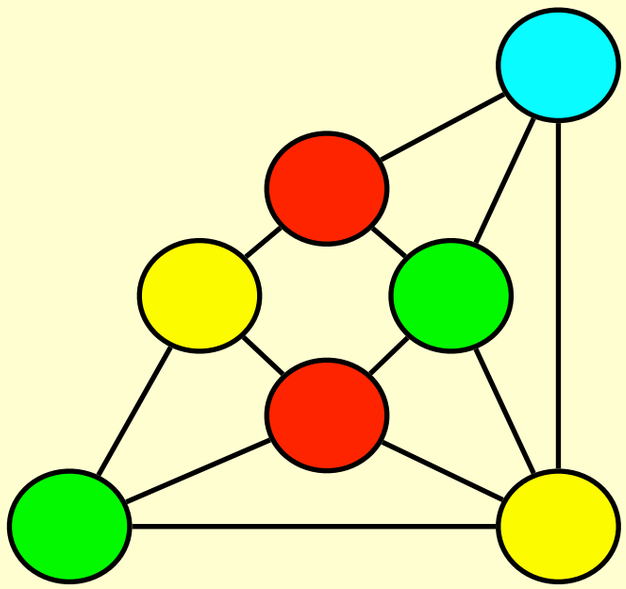
When the heuristic fails



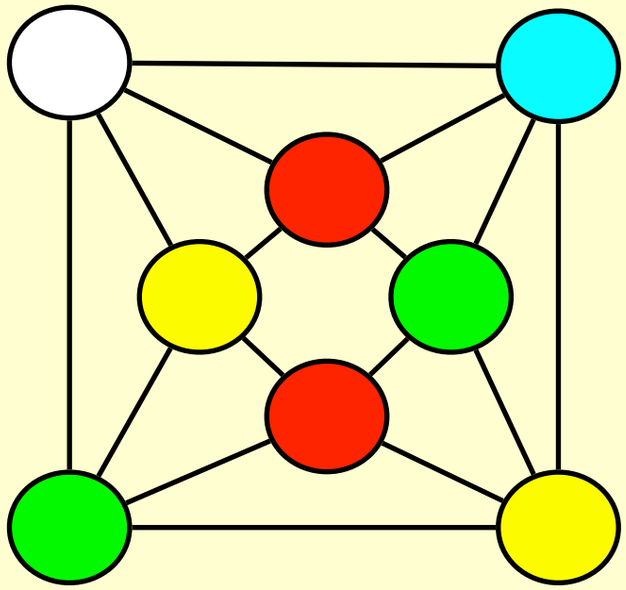
When the heuristic fails



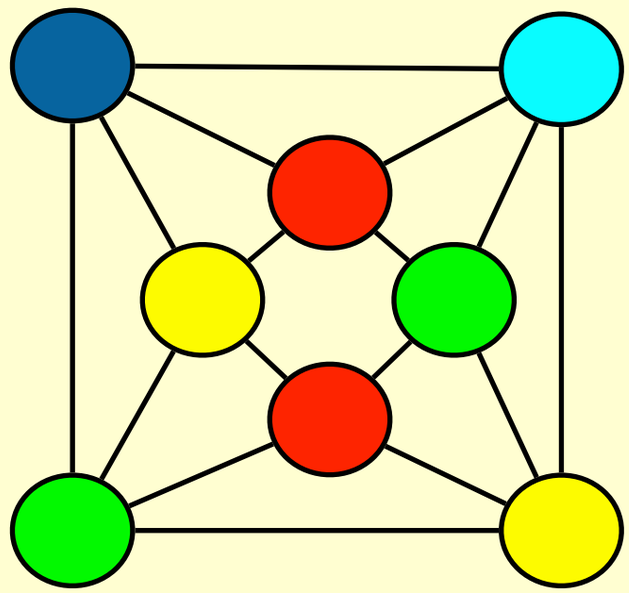
When the heuristic fails

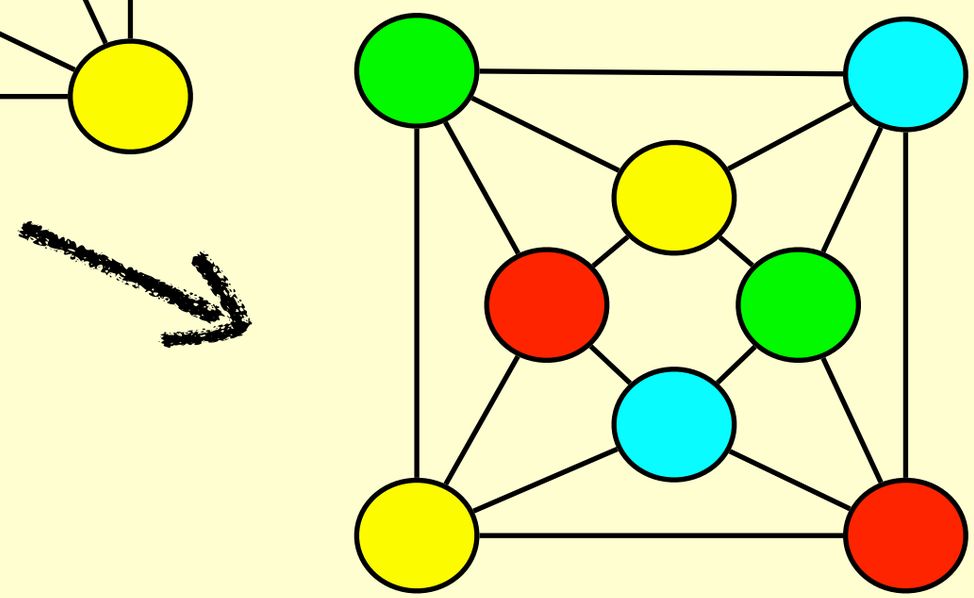


When the heuristic fails

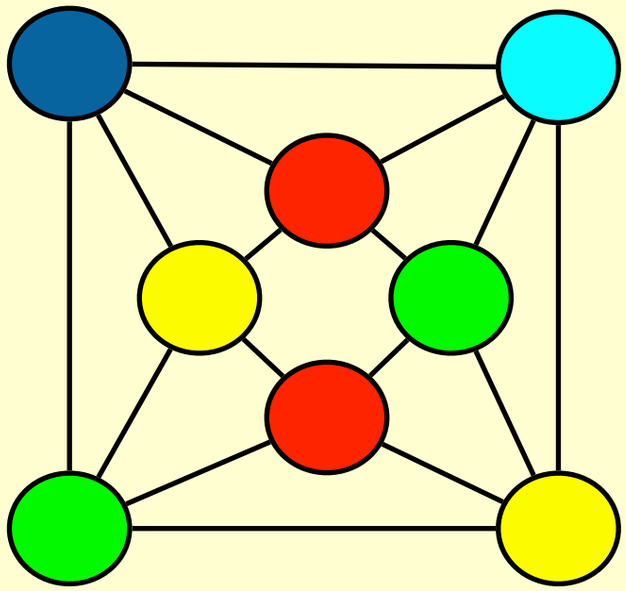


When the heuristic

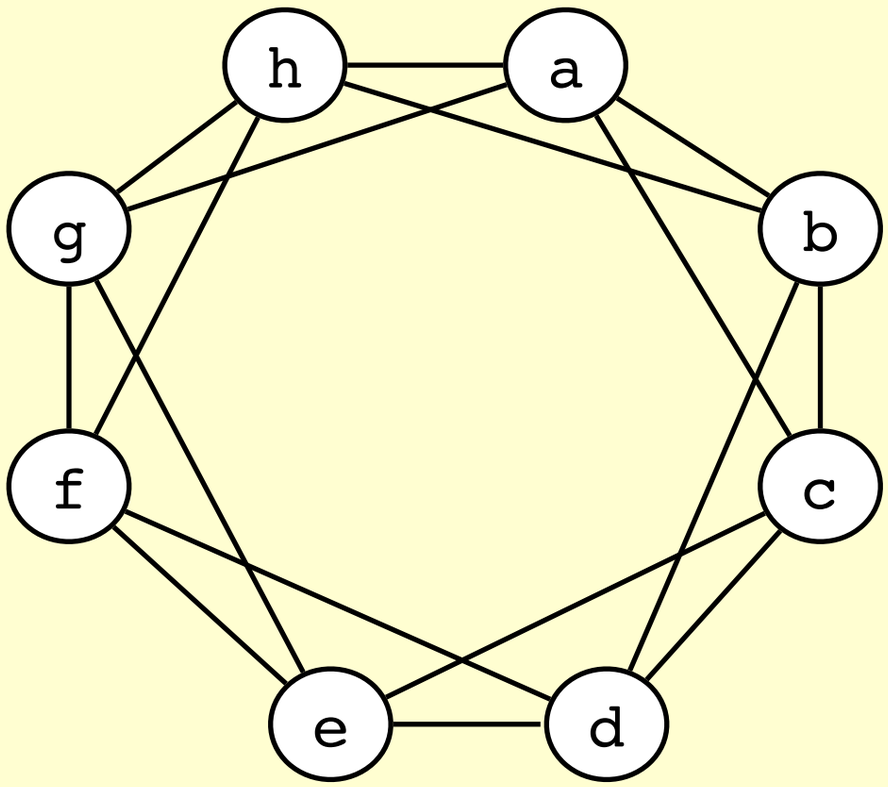




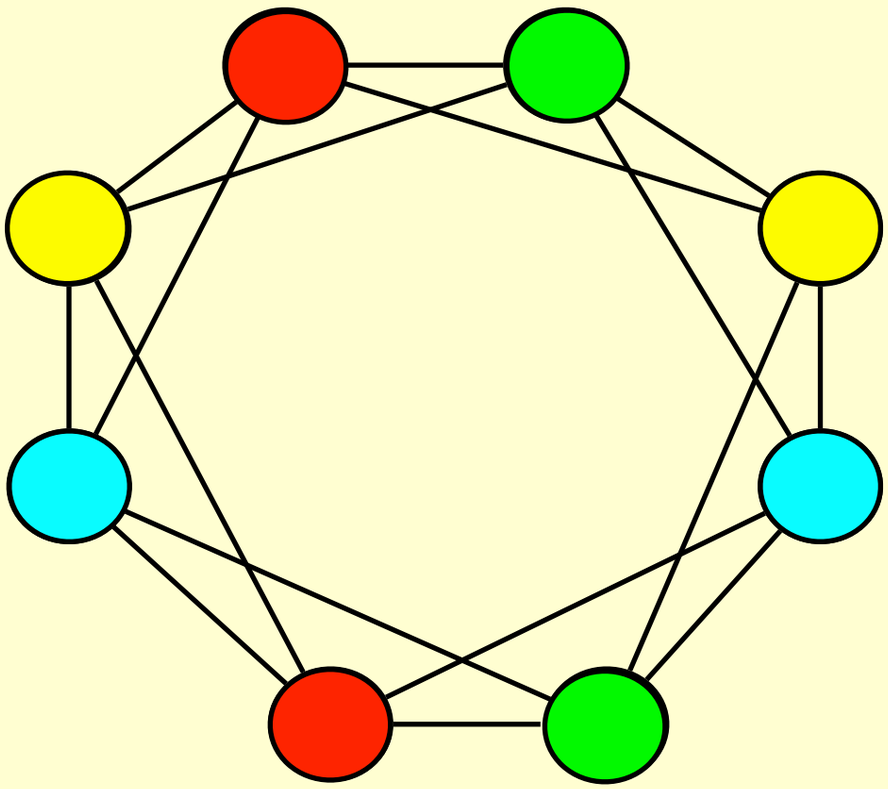
When the heuristic



When the heuristic



When the heuristic

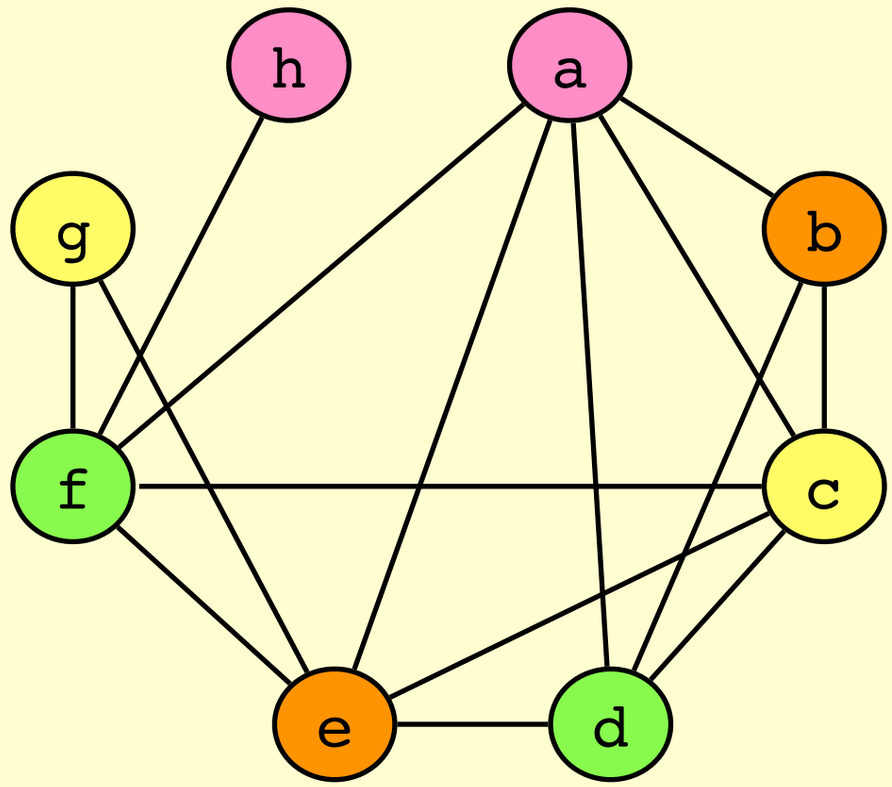


Register spilling

- Might not fit all variables into registers (either because heuristic fails, or because there are genuinely not enough registers available).
- Pick a variable to *spill* to memory, and start again.

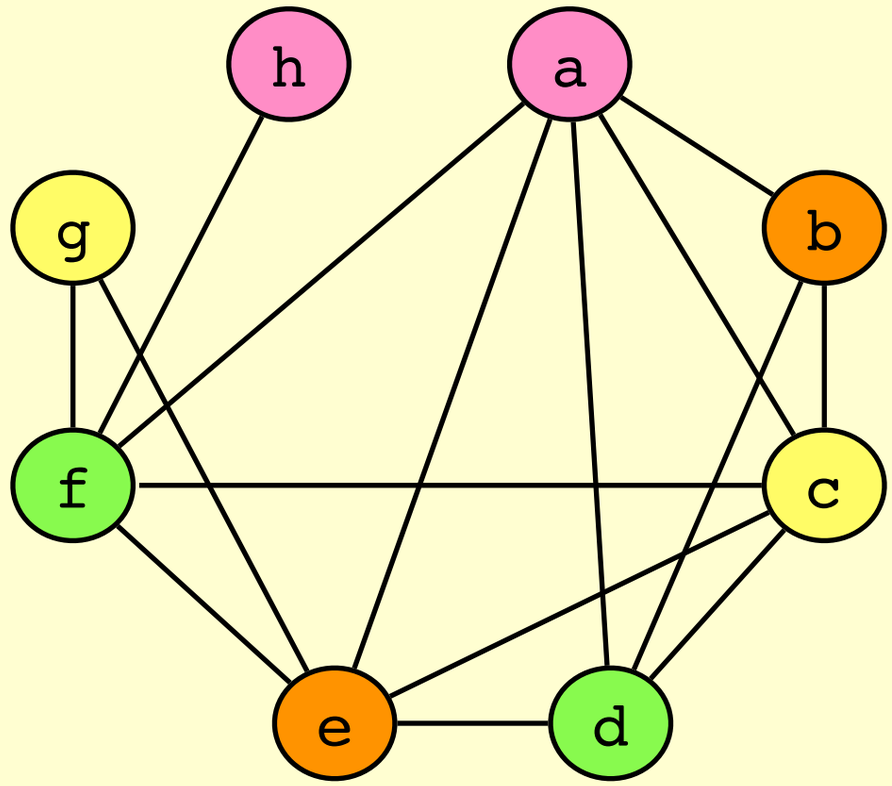
Register spilling

```
int main (int a, int b, int c) {
                        //a b c
   int d = a + b; //a b c d
   int e = d + b; //a c d e
   int f = d + a; //a c e f
   int g = a + c; //e f g
   int h = e + g; //f h
   return f + h;
}
```



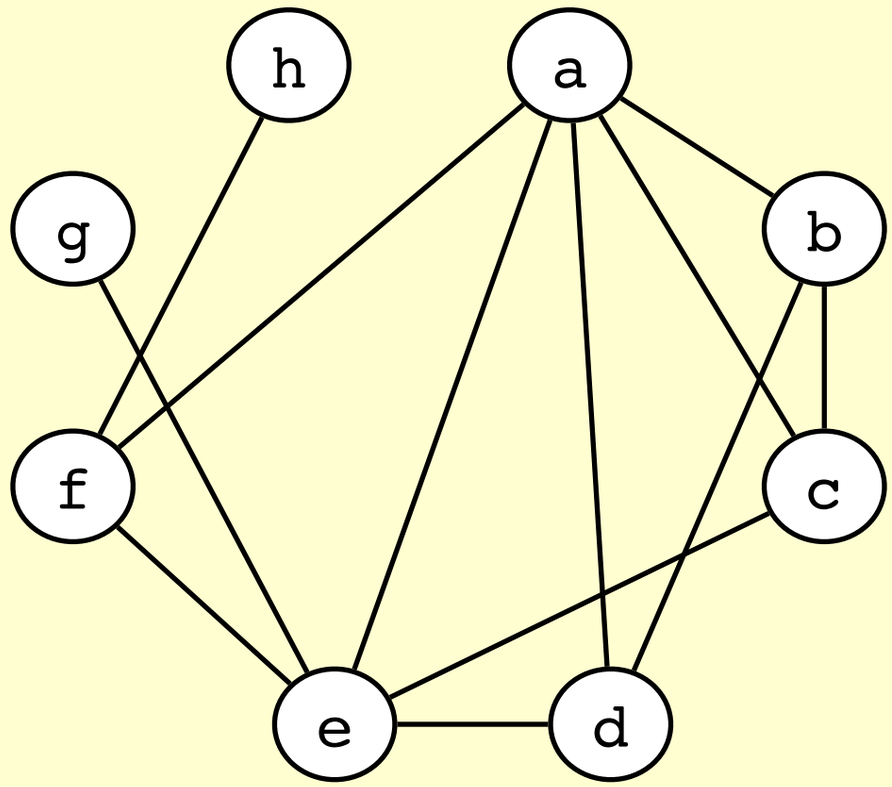
Register spilling

```
int main (int a, int b, int c) {
                        //a b c

  int d = a + b; //a b c d
  int e = d + b; //a c d e
  int f = d + a; //a c e f


  int g = a + c; //e f g
  int h = e + g; //f h

  return f + h;
}
```



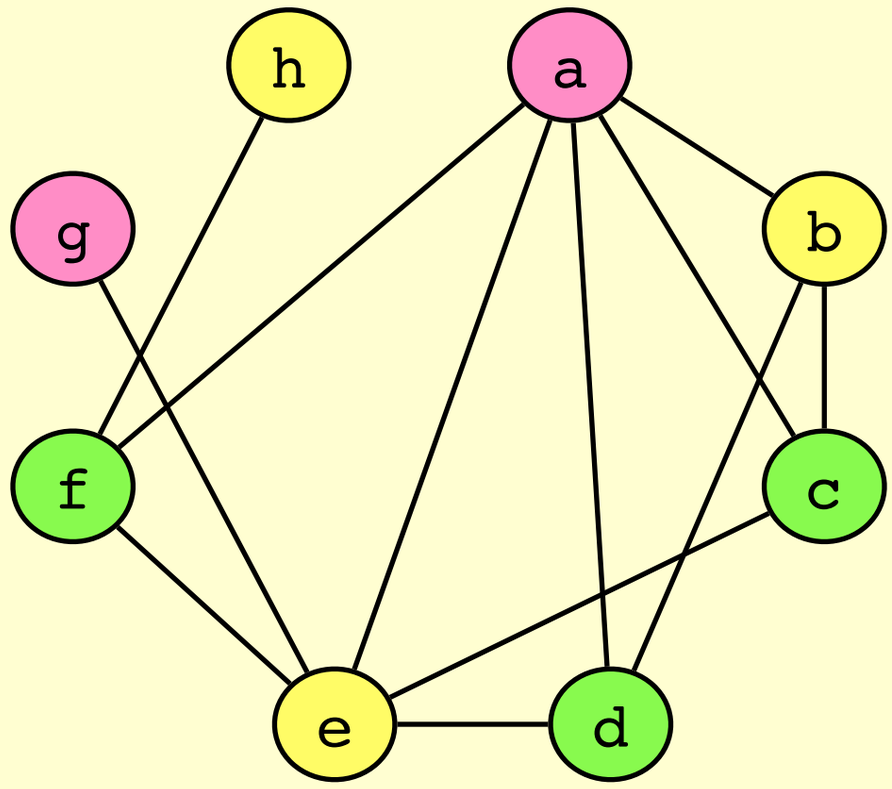
Register spilling

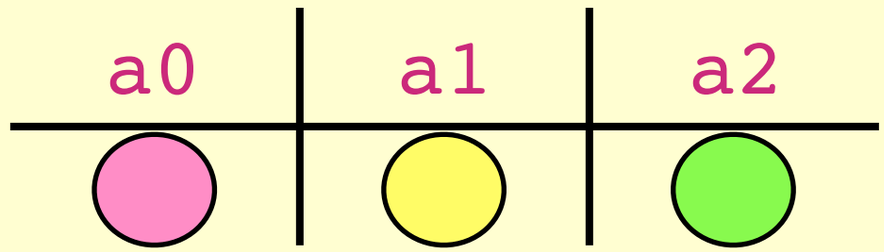
```
int main (int a, int b, int c) {
                        //a b c
  spill(c);             //a b
  int d = a + b;        //a b d
  int e = d + b;        //a d e
  int f = d + a;        //a e f
  spill(f);             //a e
  unspill(c);           //a e c
  int g = a + c;        //e g
  int h = e + g;        //h
  unspill(f);           //f h
  return f + h;
}
```



Register spilling

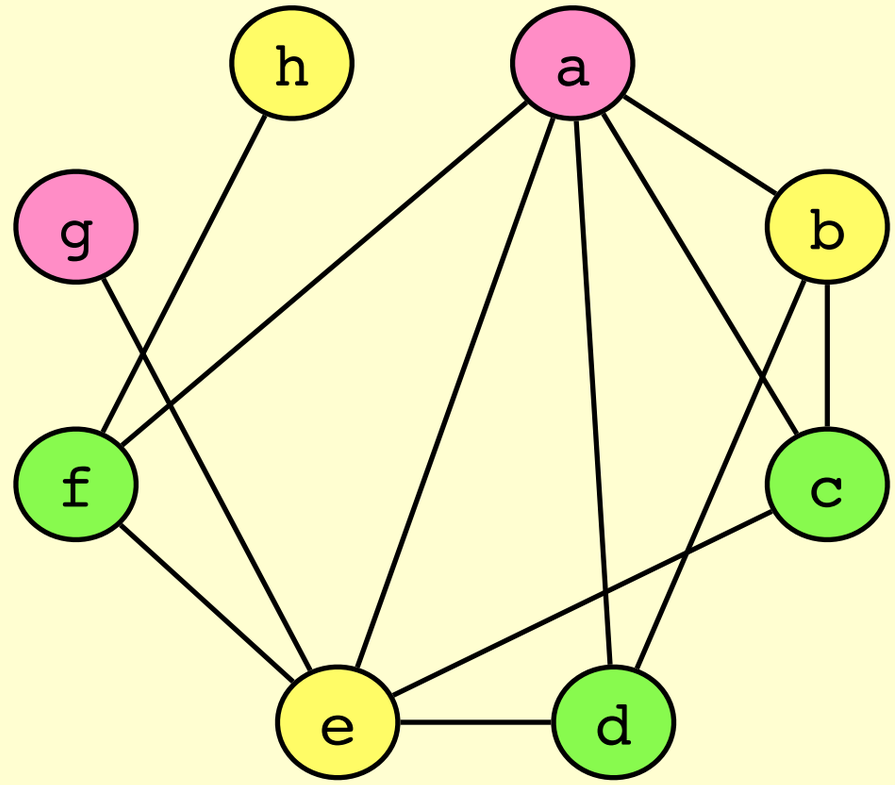
```
int main (int a, int b, int c) {
                        //a b c
  spill(c);             //a b
  int d = a + b;        //a b d
  int e = d + b;        //a d e
  int f = d + a;        //a e f
  spill(f);             //a e
  unspill(c);           //a e c
  int g = a + c;        //e g
  int h = e + g;        //h
  unspill(f);           //f h
  return f + h;
}
```

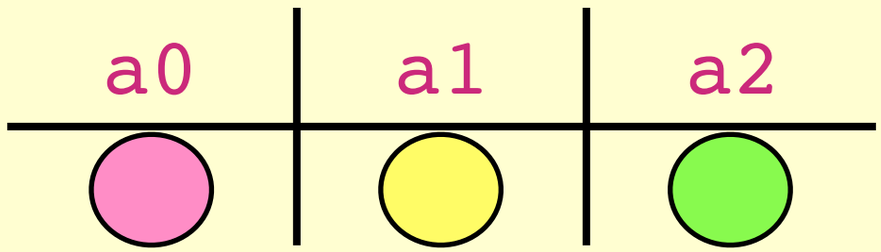




Register spilling

```
int main (a0, a1, a2) {

   SW  a2, 0(c)
   ADD a2, a0, a1
   ADD a1, a2, a1
   ADD a2, a2, a0
   SW  a2, 0(f)
   LW  a2, 0(c)
   ADD a0, a0, a2
   ADD a1, a1, a0
   LW  a2, 0(f)
   ADD a0, a2, a1
}
```





Register spilling

- Might not fit all variables into registers (either because heuristic fails, or because there are genuinely not enough registers available).

- Pick a variable to *spill* to memory, and start again.

- **Question.** Which variables would be bad candidates for spilling?

Summary

- A key component of code generation is **register allocation**.
- Register allocation can be cast as a **graph colouring** problem.
- *Optimal* graph colouring is hard – at least as hard as **SAT**.
- *Sub-optimal* graph colouring can be done with the help of **heuristics** such as "colour most-connected vertices first".
- If you run out of registers, need to **spill** some variables.