Problemi, soluzioni, codifiche

Lorenzo Repetto

repetto@calvino.ge.it

Come furono descritti gli algoritmi prima del 1945?

- Mesopotamia, ~ 40 secoli fa: i più antichi a noi noti.

  Soltanto sequenze di calcoli su particolari dati,
  non la descrizione di una procedura astratta generale.

- Grecia, ~ III secolo a.C.: Euclide, Eratostene, …

  Molti algoritmi non banali formulati in **astratto**, ma sempre informalmente, in **linguaggio naturale**.

- Nel corso dei secoli, i matematici hanno sviluppato

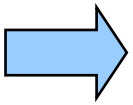
  - notazioni assai precise per descrivere la parte *statica*,
  - non altrettanto per la parte *dinamica*; in particolare:

    invece di $\mathbf{x \leftarrow x + 3}$ sempre $\mathbf{x_{n+1} = x_n + 3}$

Nel diagramma originale stilato da Ada Byron (1843):

• manca l'idea di *array*, sicché dev'essere preordinata una (lunga e precisa) sequenza di *variable-card*, per ripetere le operazioni su variabili "con indice aumentato di un'unità" di volta in volta;
• non sono formalizzate le istruzioni di salto condizionato, ...
ma tutto è descritto precisamente!
• è impostato un calcolo di complessità (numero di operazioni aritmetiche), ma nulla è detto sui progressivi errori numerici.

Punti di forza delle sue *Note*:

• capacità di sintesi e di penetrazione nelle problematiche epistemologiche;
• la macchina analitica è potenzialmente universale: permette qualsiasi computazione (già intuito da Babbage);
• i numeri possono rappresentare entità che non siano mere quantità o misure  calcolo simbolico !

Circa un secolo dopo…

Notazione veramente precisa e completa,
"a livello macchina", sia in stile funzionale (λ-calcolo di Church) sia in stile imperativo (macchine di Turing)

Definizioni di funzione (effettivamente) calcolabile e di algoritmo: diversi formalismi, ma stessa nozione!

- calcolo dei combinatori – Schönfinkel e Curry, 1920-1930
- λ-calcolo – Church, Rosser e Kleene, 1931-1941
- ricorsività generale – Gödel, 1934 (da Herbrand)
- calcolo delle equazioni – Kleene, 1936 (da Gödel)
- macchine (universali) di Turing – 1936
- sistema di produzioni canonico di Post – 1936-1947

“Calcolo delle equazioni”

Definizioni intensionali di funzioni con proprio nome, da *n*-uple di naturali ai naturali, applicabili anche ai risultati di (altre) funzioni, in generale parziali, ricorsive o mutuamente ricorsive.
Esempio: la funzione di Ackermann (1928), ricorsiva totale ma non ricorsiva primitiva.

$$\mathrm{A}(0, y) = y + 1$$
$$\mathrm{A}(x + 1, 0) = \mathrm{A}(x, 1)$$
$$\mathrm{A}(x + 1, y + 1) = \mathrm{A}(x, \mathrm{A}(x + 1, y))$$

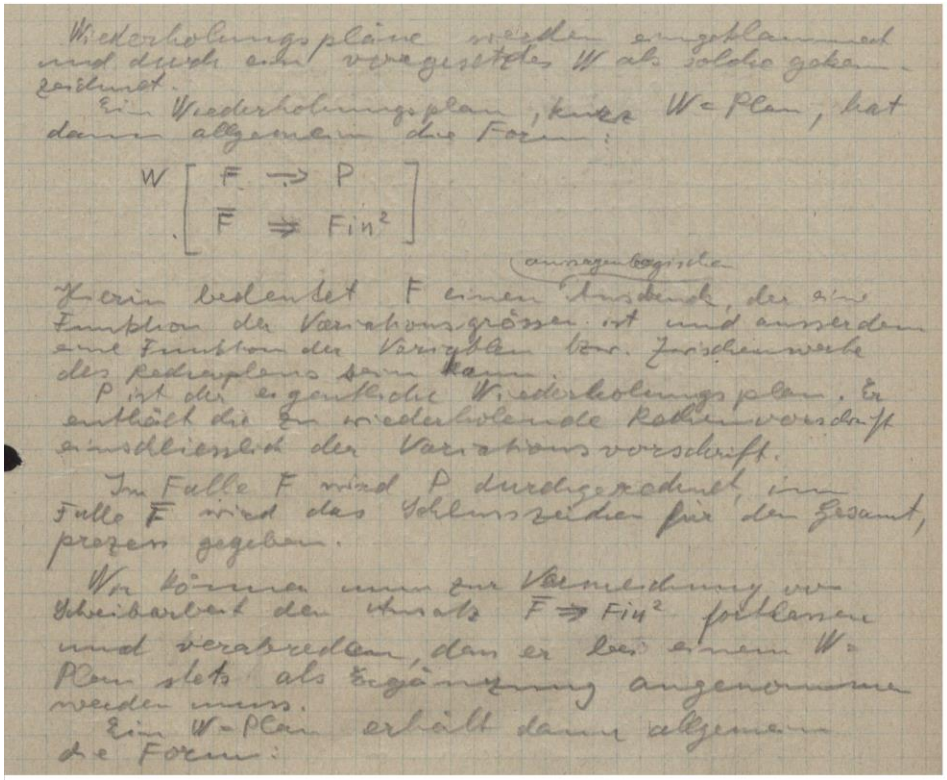
In ML (1973):

```
fun A(x, y) =
      if x=0 then y+1
      else if y=0 then A(x-1, 1)
           else A(x-1, A(x, y-1));
```

Comandi / istruzioni nel Plankalkül: “Lo scopo del Plankalkül è fornire una descrizione puramente formale di qualsiasi processo computazionale.”

- assegnazione: **x + 3 ⇒ x**
  Zuse fu il primo… ed era consapevole di ciò che faceva!
  ⇒ Ergibt-Zeichen = segno di produzione, con ricevente a destra; Rutishauser lo propose per l’ALGOL 58 …
- condizionale: una sorta di “guarded command” $F_i \rightarrow P_i$ (Bedingt-Zeichen)
- varie forme di cicli (anche annidati):
  - a numero di iterazioni precalcolato
  - oppure no, con “guardie” riprese dalla logica dei predicati del prim’ordine (anche con **quantificatori**), operatore **μ** …
- niente goto (né label), ma possibilità di interrompere una sequenza di istruzioni ed eventualmente uscire da un ciclo al livello di *nesting* specificato – quindi il goto non serve!
- niente primitive per I/O – dipendenti dalla macchina!

Wiederholungspläne (Zuse, manoscritto originale)

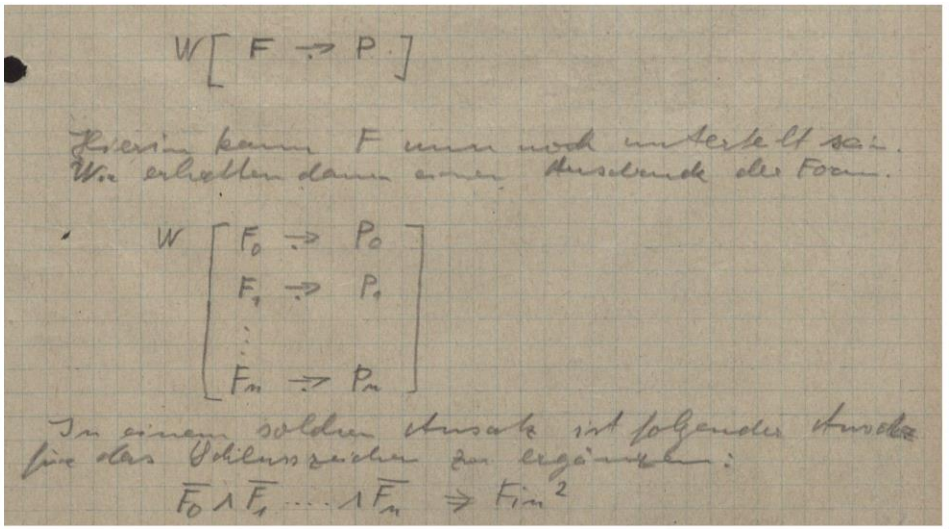


Wiederholungspläne werden eingeklammert und durch ein vorangestelltes W als solche gekennzeichnet.
Ein Wiederholungsplan, kurz W-Plan, hat dann allgemein die Form:

$$W\left[\begin{array}{l} F \rightarrow P \\ \overline{F} \Rightarrow Fin^2 \end{array}\right]$$

(anzugeben logisch)

F kann bedeutet F einen Ausdruck der eine Funktion der Variablenmengen ist und ausserdem eine Funktion der Variablen bzw. Zwischenwerte des Rechenplanes sein kann.
P ist die eigentliche Wiederholungsplan. Er enthält die zu wiederholende Rechenvorschrift einschliesslich der Variablenvorschrift.
Im Falle F wird P durchgerechnet, im Falle $\overline{F}$ wird das Schlusszeichen für den Gesamtprozess gegeben.
Wir können nun eine Vereinbarung vereinbaren den Ausdruck $\overline{F} \Rightarrow Fin^2$ fortlassen und verabreden, dass er bei einem W-Plan stets als Ergänzung angenommen werden muss.
Ein W-Plan erhält dann allgemein die Form:



$$W\left[F \rightarrow P.\right]$$

Hierin kann F nun noch unterteilt sein. Wir erhalten dann einen Ausdruck der Form.

$$W\left[\begin{array}{l} F_0 \rightarrow P_0 \\ F_1 \rightarrow P_1 \\ \vdots \\ F_n \rightarrow P_n \end{array}\right]$$

In einem solchen Ausdruck ist folgender Ausdruck für das Schlusszeichen zu ergänzen:

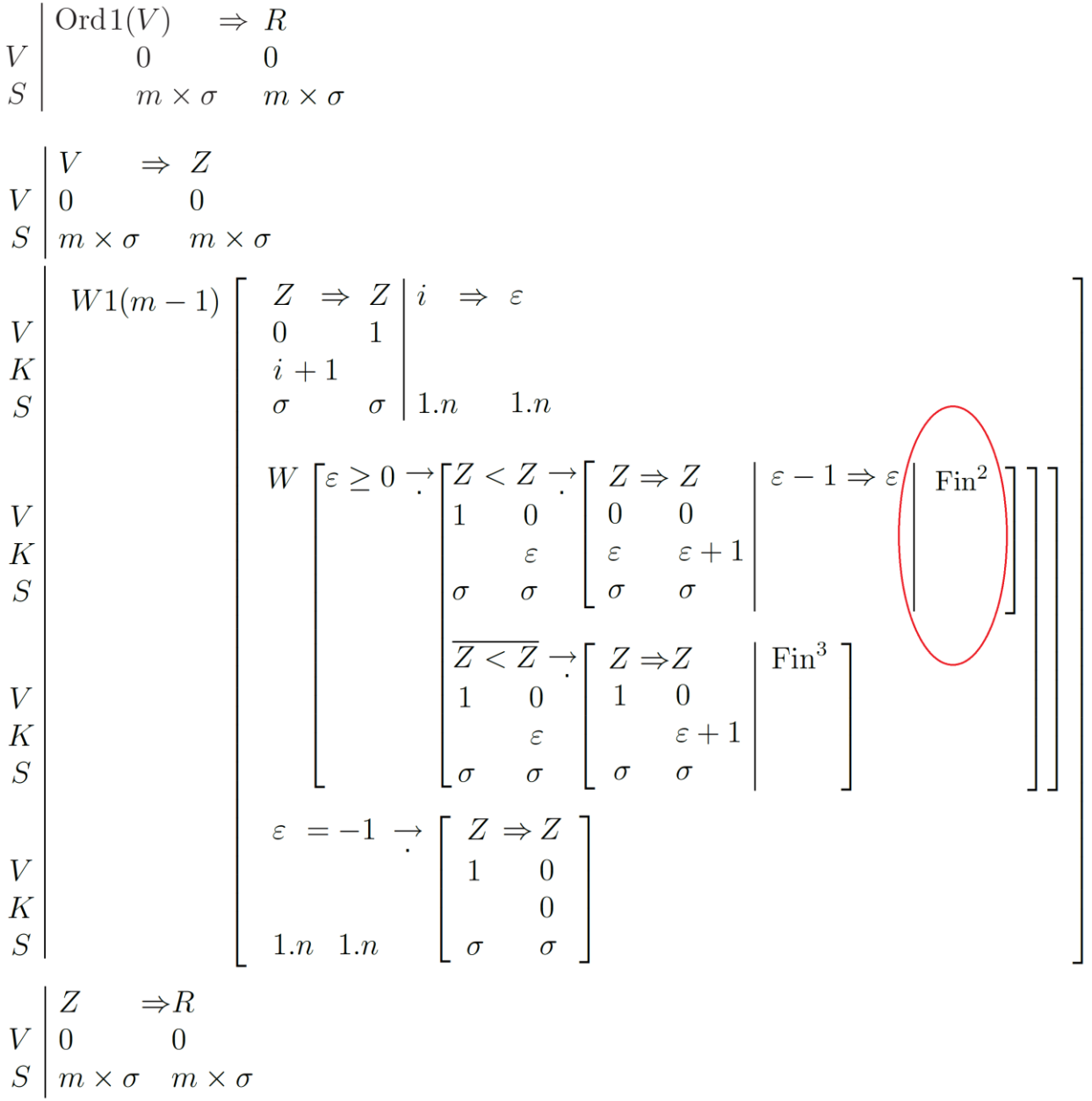
$$\overline{F_0} \wedge \overline{F_1} \cdots \wedge \overline{F_n} \Rightarrow Fin^2$$

d.h. wenn für keinen der Teilpläne die Bedingung zur Durchrechnung gegeben ist, wird der Prozess abgebrochen.

$F_i \overset{\rightarrow}{.} P_i$ è un'istruzione, equivalente a if_then_ (ma senza else_)

Nota: qui Fin^2 è trattata come una variabile…

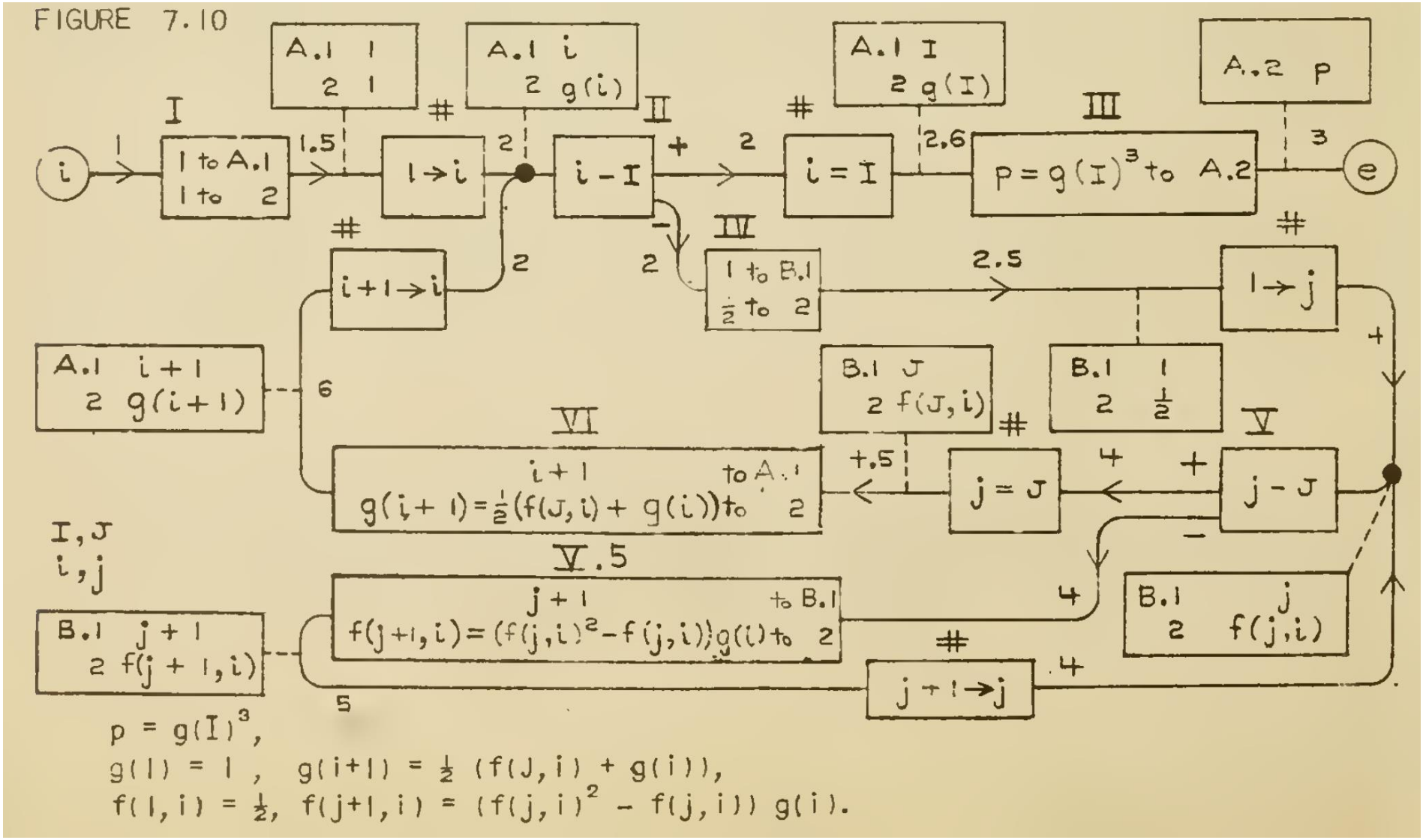
Ordinare un array con *straight insertion sort* (P3.27)



N.B.: Fin^2 e Fin^3

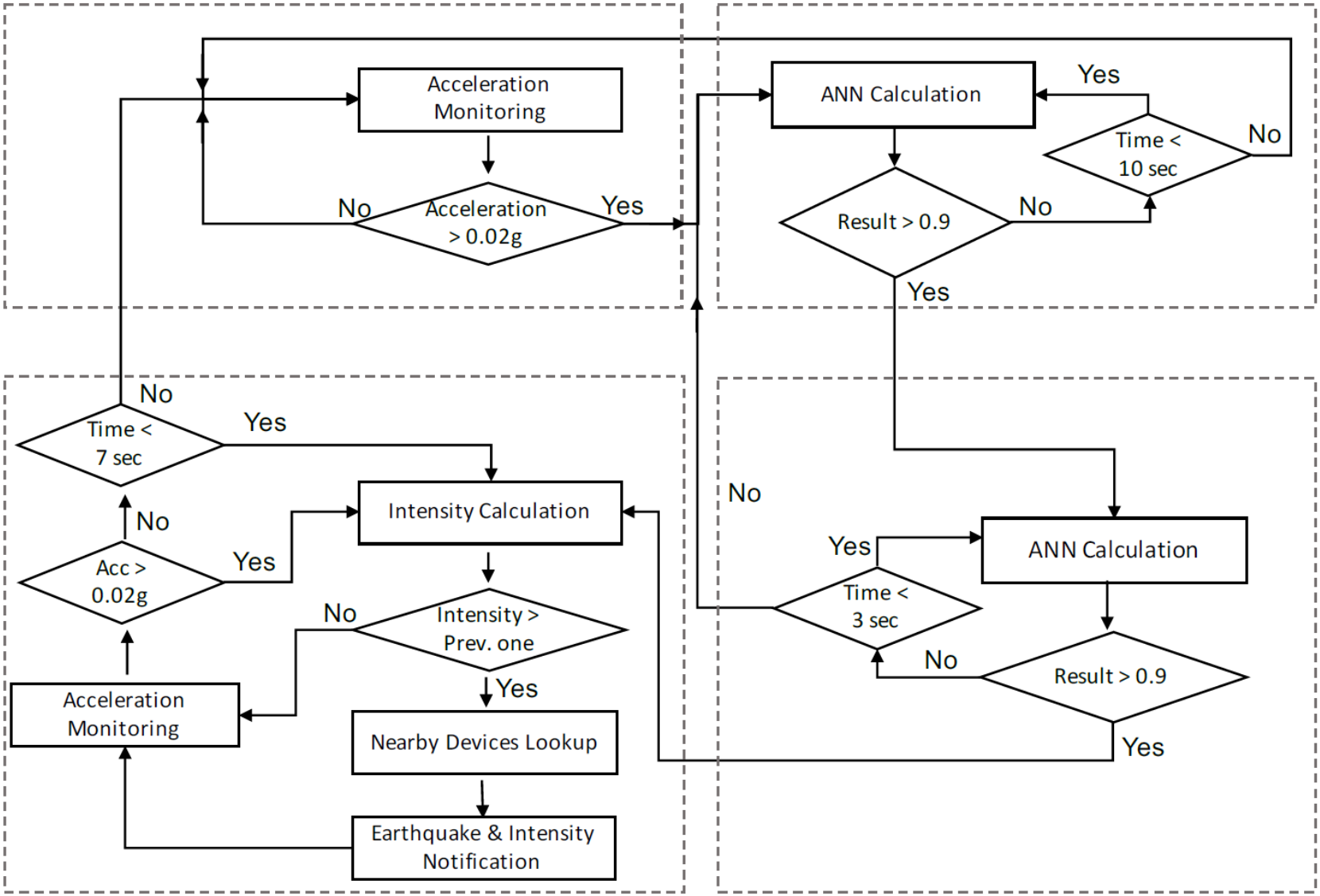
NON Fin e Fin^2 perché c'è anche il livello della guardia interna!

(Ho aggiunto Fin^2, altrimenti la condizione successiva può far riferimento all'elemento di posto –1, inesistente…)



… Diagrammi per pianificare i programmi …

(H. Goldstine e J. von Neumann, 1947)



… A flow chart to detect an earthquake …

(J. Lee *et alii*, 2019)

- Condizioni per ottenere un risultato da una procedura di calcolo
- Provarne la correttezza …

L'attività di codifica di una procedura di calcolo «has to be viewed as a logical problem and one that represents a new branch of formal logics.»

(H. H. Goldstine & J. von Neumann, 1947)

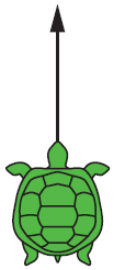
«The programmer should make assertions about the various states that the machine can reach. The checker has to verify that [these assertions] agree with the claims that are made for the routine as a whole. Finally the checker has to verify that the process comes to an end.»

(A. M. Turing, 1949)

Da una proposta per le ultime classi di una scuola primaria:

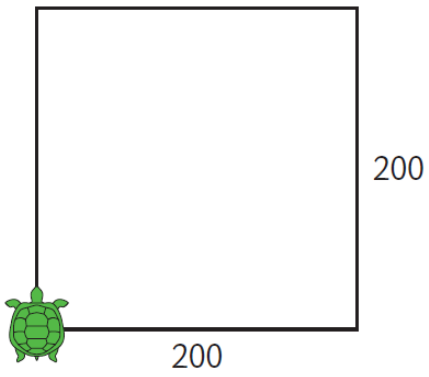
[…] L'informatica a scuola spesso si limita all'uso di computer, tablet e programmi applicativi. Tuttavia il contributo culturale più significativo che ci offre l'informatica è il "pensiero computazionale", ovvero l'insieme dei processi mentali che mette in atto un informatico nella sua tipica attività di *problem solving*. Si tratta di competenze trasversali, utili e declinabili in tutti gli ambiti disciplinari: formulare i problemi in modo che possano essere risolti in maniera automatica da agenti autonomi, analizzare e organizzare con logica le informazioni, rappresentarle attraverso modelli e astrazioni, automatizzare lo svolgimento di compiti tramite sequenze di passi ordinati, generalizzare e trasferire processi risolutivi a una grande varietà di situazioni diverse. […]

Primi comandi in LOGO https://xlogo.inf.ethz.ch/release/latest/

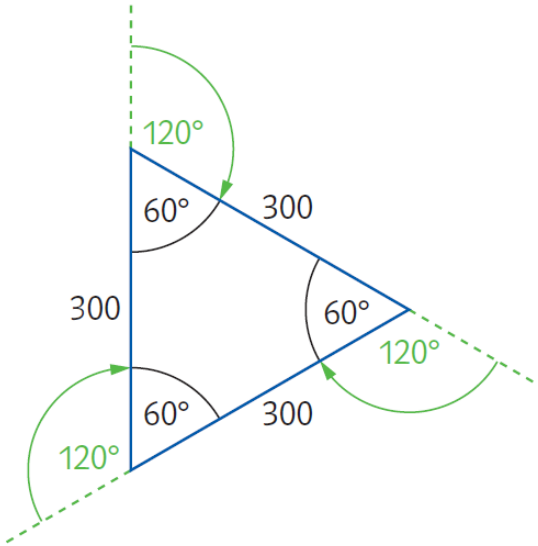


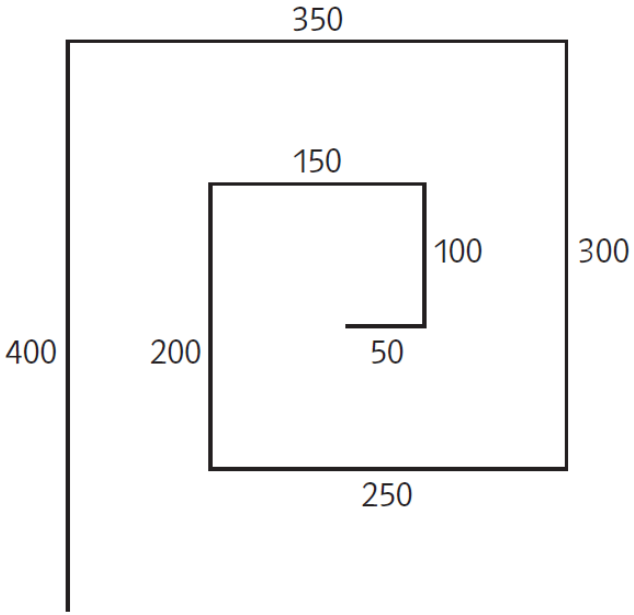
posizione
e direzione
iniziali

`fd` **numero**	vai in avanti per un dato numero di passi
`bk` **numero**	vai indietro per un dato numero di passi
`rt` **numero**	gira di un dato angolo (in gradi) a destra
`lt` **numero**	gira di un dato angolo (in gradi) a sinistra
`setpc` **colore**	cambia il colore della penna
`cs`	cancella tutti i tratti disegnati e torna alla posizione iniziale

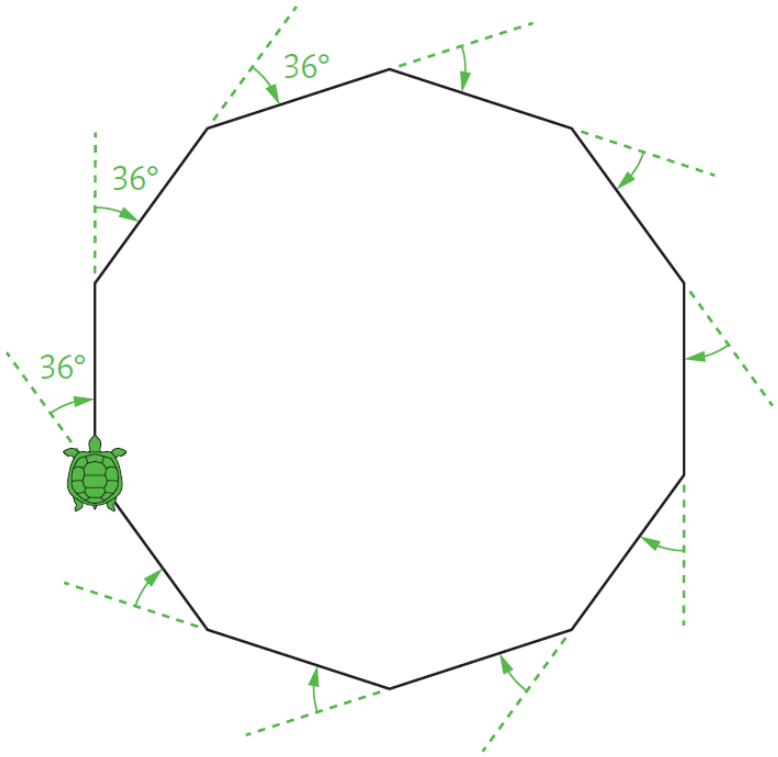



```
repeat 4 [fd 200 rt 90]
```



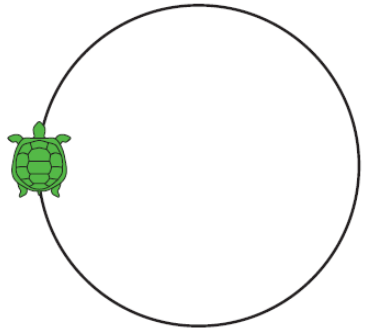


Poligoni “regolari” e linee “curve”




```
repeat 10 [fd 100 rt 36]
```

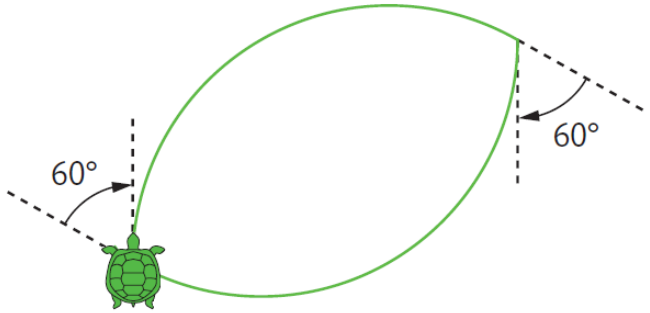
```
repeat m [fd n rt 360/m]
```



?

```
repeat m [fd c/m rt 360/m]
```

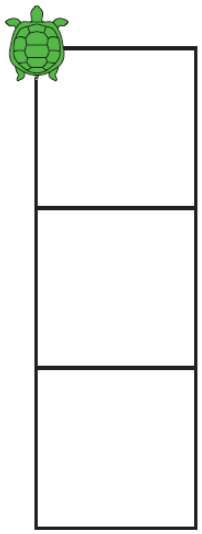
m = 360 o m = 180 , c multiplo di m




```
setpc green
repeat 2 [repeat 120 [fd 3 rt 1] rt 60]
```

Nominare programmi, definire uno o più parametri

```
1  to quadrato100
2  repeat 4 [fd 100 rt 90]
3  end

repeat 3 [quadrato100  fd 100]
```



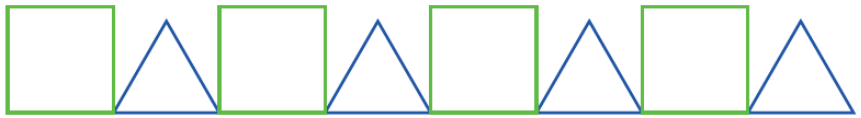
```
1  to quadrato :lato
2  repeat 4 [fd :lato rt 90]
3  end

repeat 3 [ quadrato 100  fd 100 ]
```

```
1  to poligono :gono :lato :colore
2  setpc :colore repeat :gono [fd :lato rt 360/:gono]
3  end

repeat 3 [ poligono 4 100 black  fd 100 ]
```

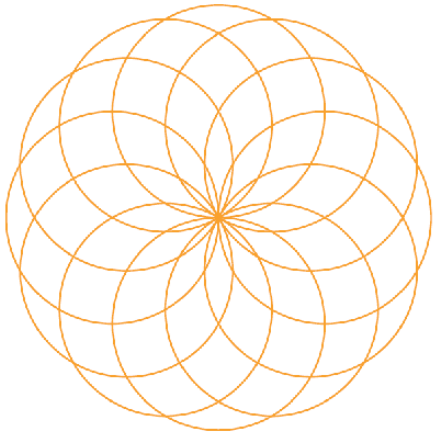
Alcuni disegni, usando la procedura “poligono”

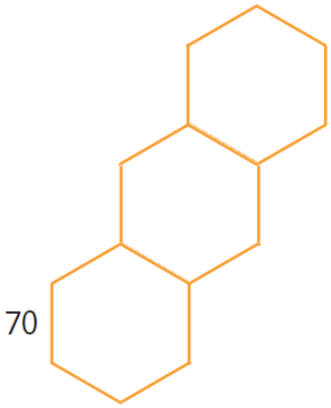
```
1  to poligono :gono :lato :colore
2  if :gono > 2 [ setpc :colore repeat :gono [fd :lato rt 360/:gono] ]
3  end
```

```
repeat 4 [
  poligono 4 100 green
  rt 90 fd 100 lt 60
  poligono 3 100 blue
  rt 60 fd 100 lt 90
]
```



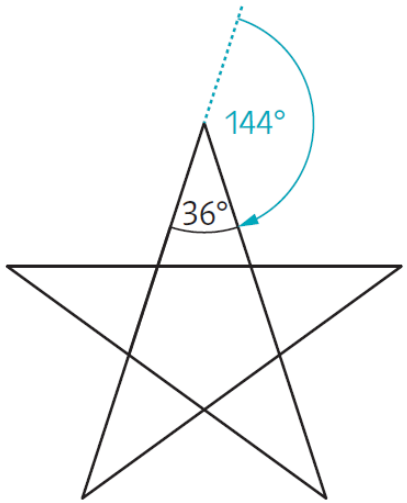
```
repeat 12 [
  poligono 360 2 orange  rt 30
  wait 10
]
```



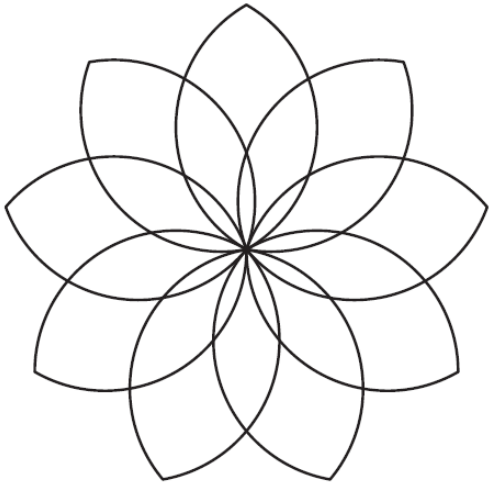



```
repeat 3 [
  poligono 6 70 orange
  fd 70 rt 60 fd 70 lt 60
]
```

Una stella a 5 punte e una corolla con 9 petali

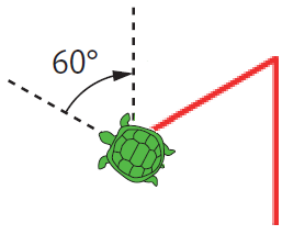



```
rt 18  repeat 5 [ fd 200  rt 144 ]
```

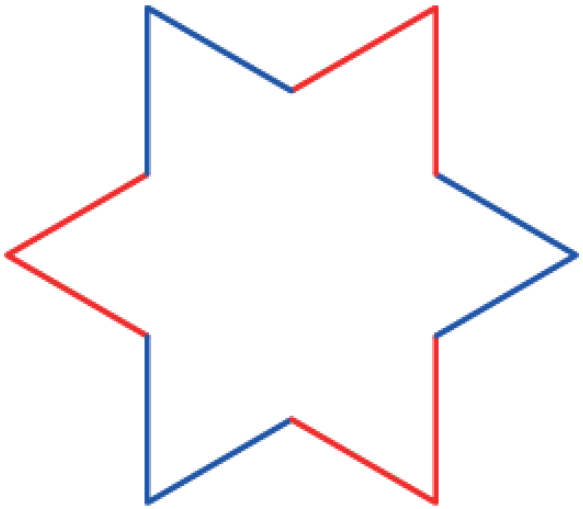


```
to terzocirc
repeat 120 [ fd 3  rt 1 ]
end

to petalo
repeat 2 [ terzocirc  rt 60 ]
end

to corolla
repeat 9 [ petalo  rt 40 ]
end

corolla
```

Una stella a 6 punte “animata”




```
to punta
fd 100 lt 120 fd 100 rt 60
end

setpc red
punta
```

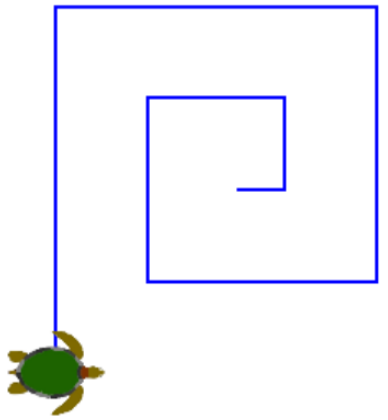


```
to stella
repeat 3 [setpc red punta setpc blue punta]
end

to stella_animata
ht stella
repeat 60 [
  wait 10
  setpc white
  repeat 6 [ punta ]
  rt 6
  stella
] st
end

stella_animata
```

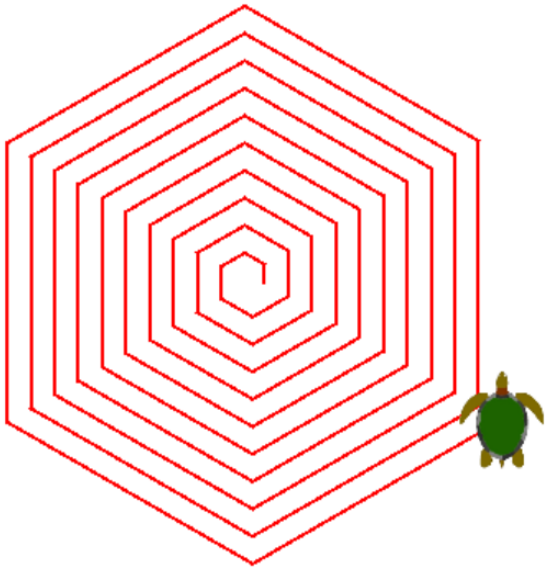
`ht`	nascondi la tartaruga
`st`	mostra la tartaruga

Spirali in forma “ricorsiva”

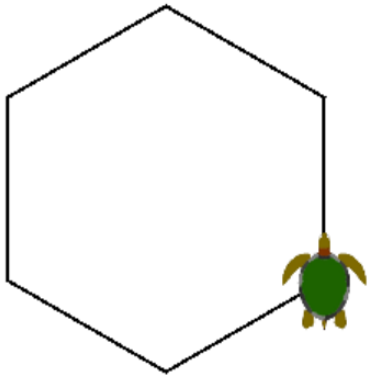
```
to spirale_inf :lato :ang :inc
fd :lato lt :ang
spirale_inf :lato+:inc :ang :inc
end

to spirale_ric :lato :ang :inc :n
if :n > 0 [
  fd :lato lt :ang
  spirale_ric :lato+:inc :ang :inc :n-1
]
end
```



```
rt 90
spirale_ric 50 90 50 8
```



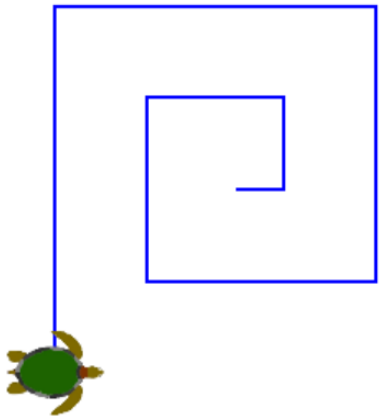
spirale_ric 20 60 5 60



spirale_ric 200 60 0 6

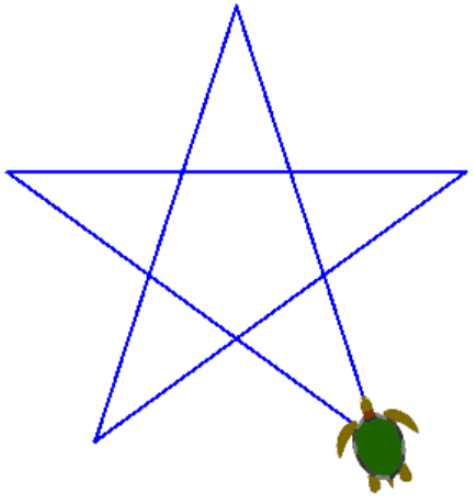
Spirali in forma “iterativa” (equivalente)

```
to spirale_it :lato :ang :inc :n
make "lato_cor :lato
repeat :n [
  fd :lato_cor lt :ang
  make "lato_cor :lato_cor+:inc
]
end
```

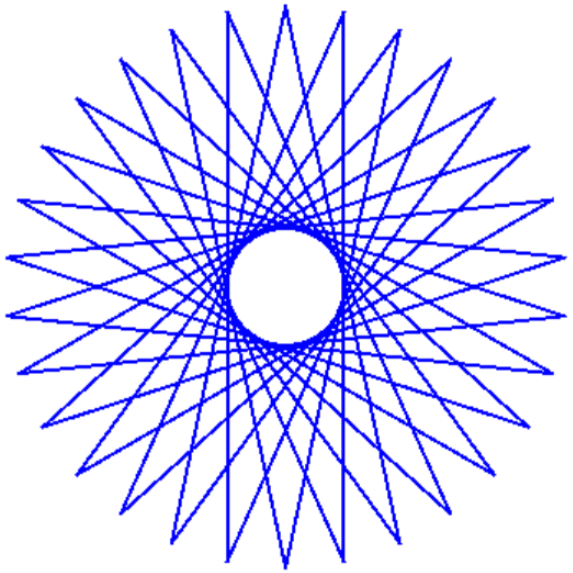


```
rt 90
spirale_it 50 90 50 8
```

Spirali?

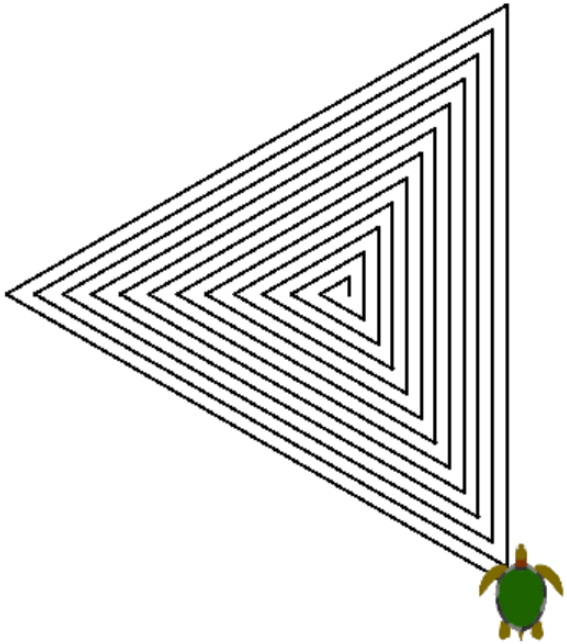


```
lt 18
spirale_it 200 144 0 5
```

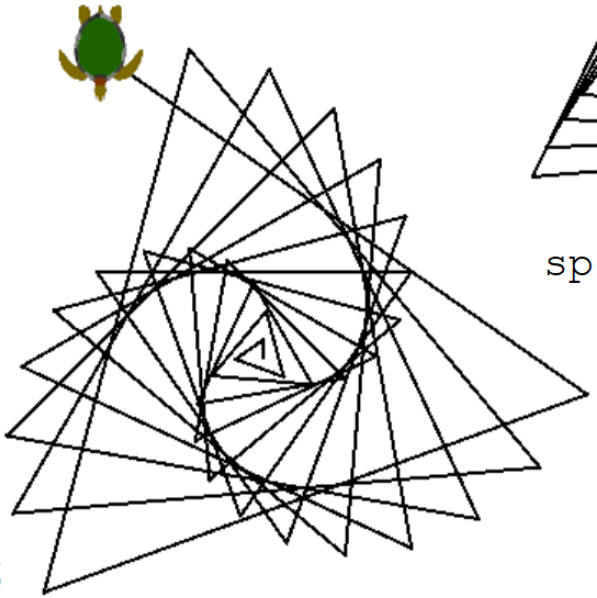


```
spirale_it 200 156 0 30
```

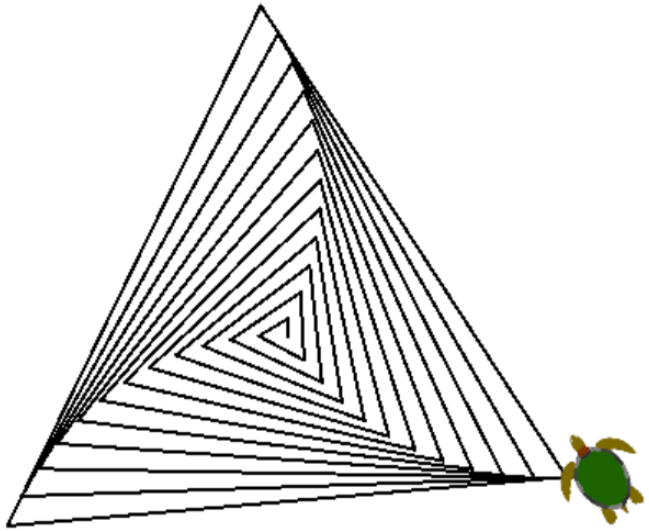
Altre spirali …



spirale_it 20 **120** 18 36



spirale_it 20 **125** 18 36



spirale_it 20 **121** 18 36

Provare i comandi:

spirale_inf 15 **121** 15

spirale_inf 15 **125** 15

Colorare superfici

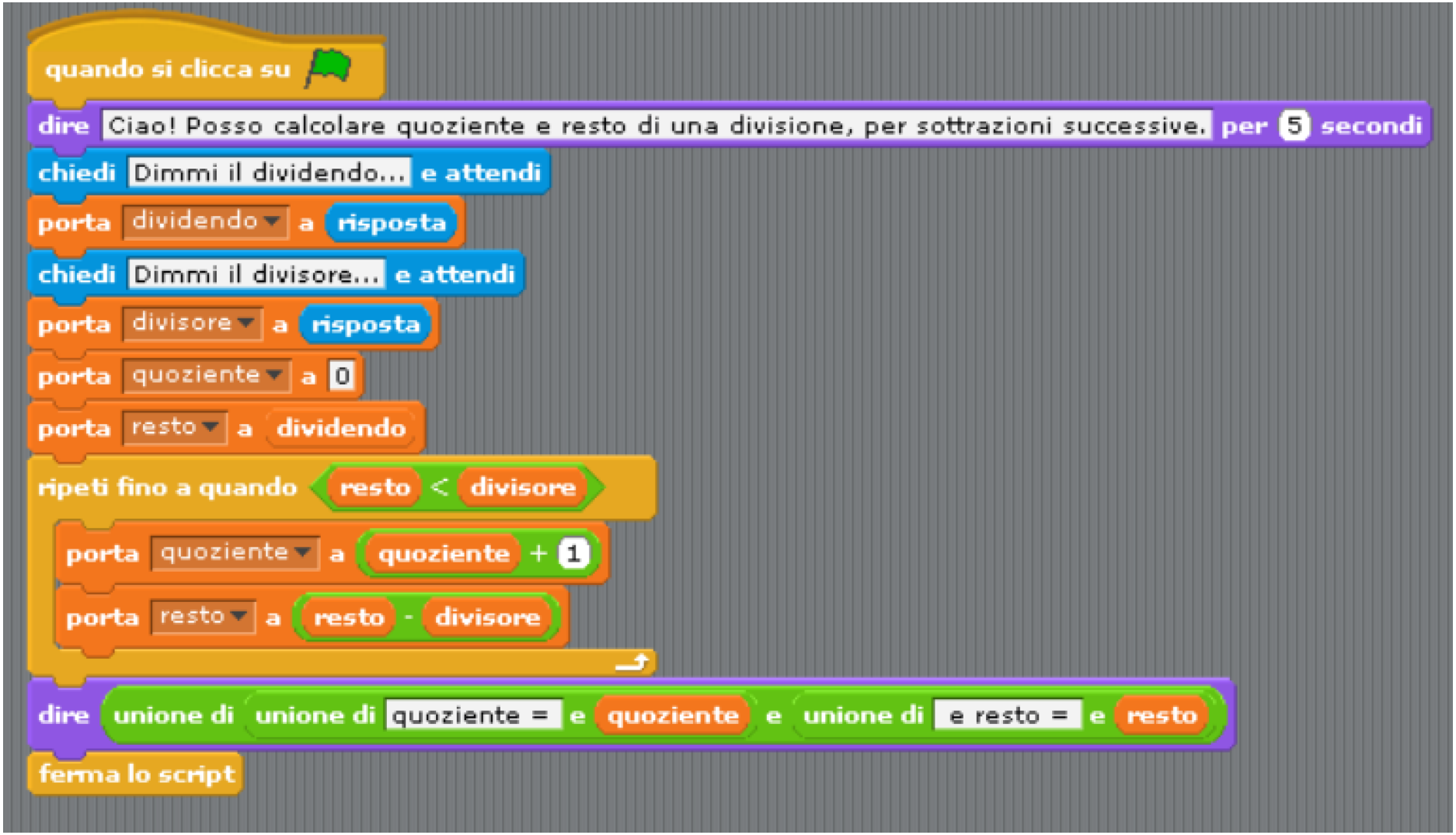

```
1  to quadratopieno :lato :colore
2  setpc :colore
3  repeat :lato/2 [fd :lato rt 90 fd 1 rt 90 fd :lato lt 90 fd 1 lt 90]
4  end
```

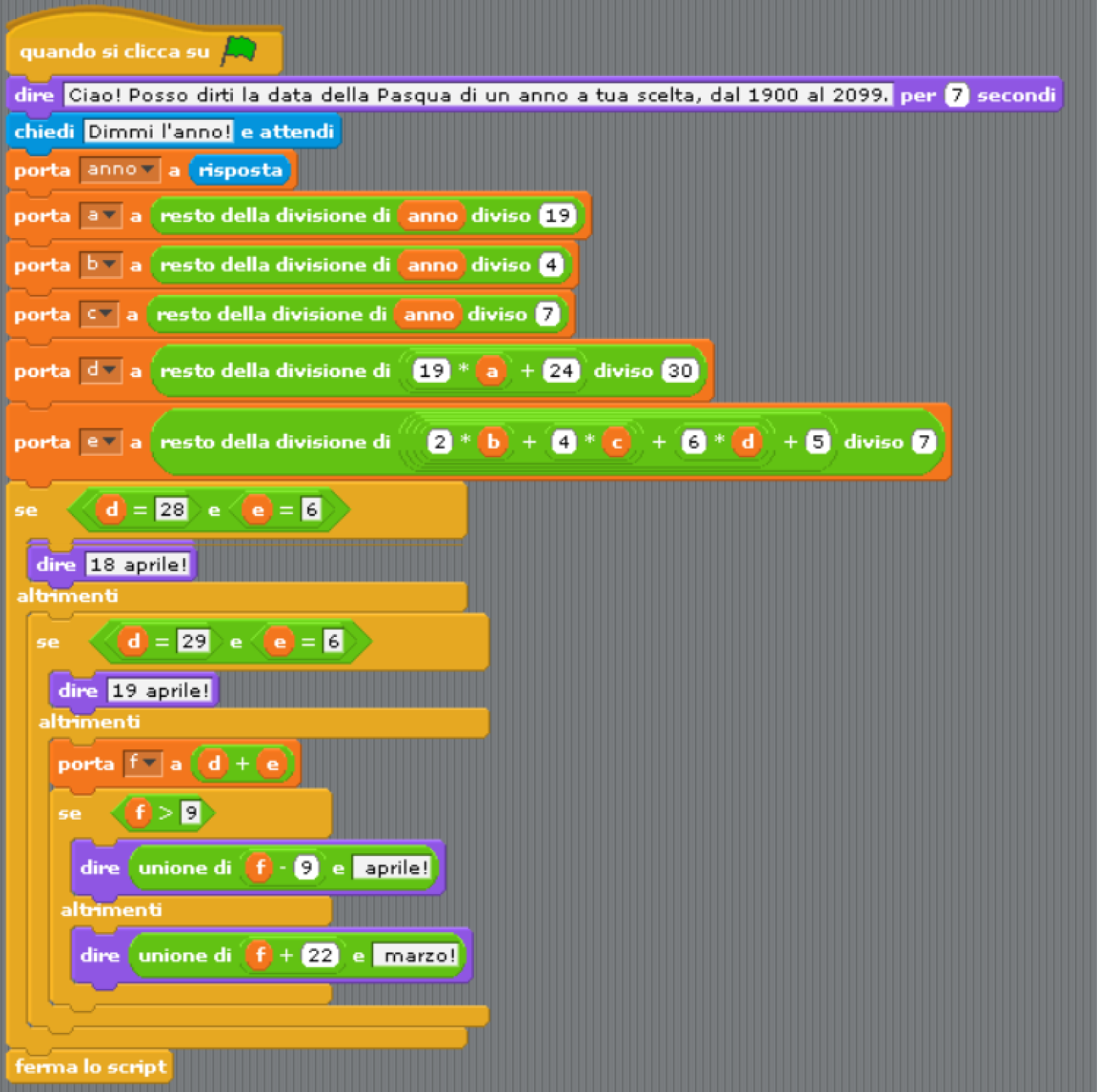
```
setpc white lt 90 fd 400 rt 90
repeat 4 [ quadratopieno 100 yellow  quadratopieno 100 green ]
```

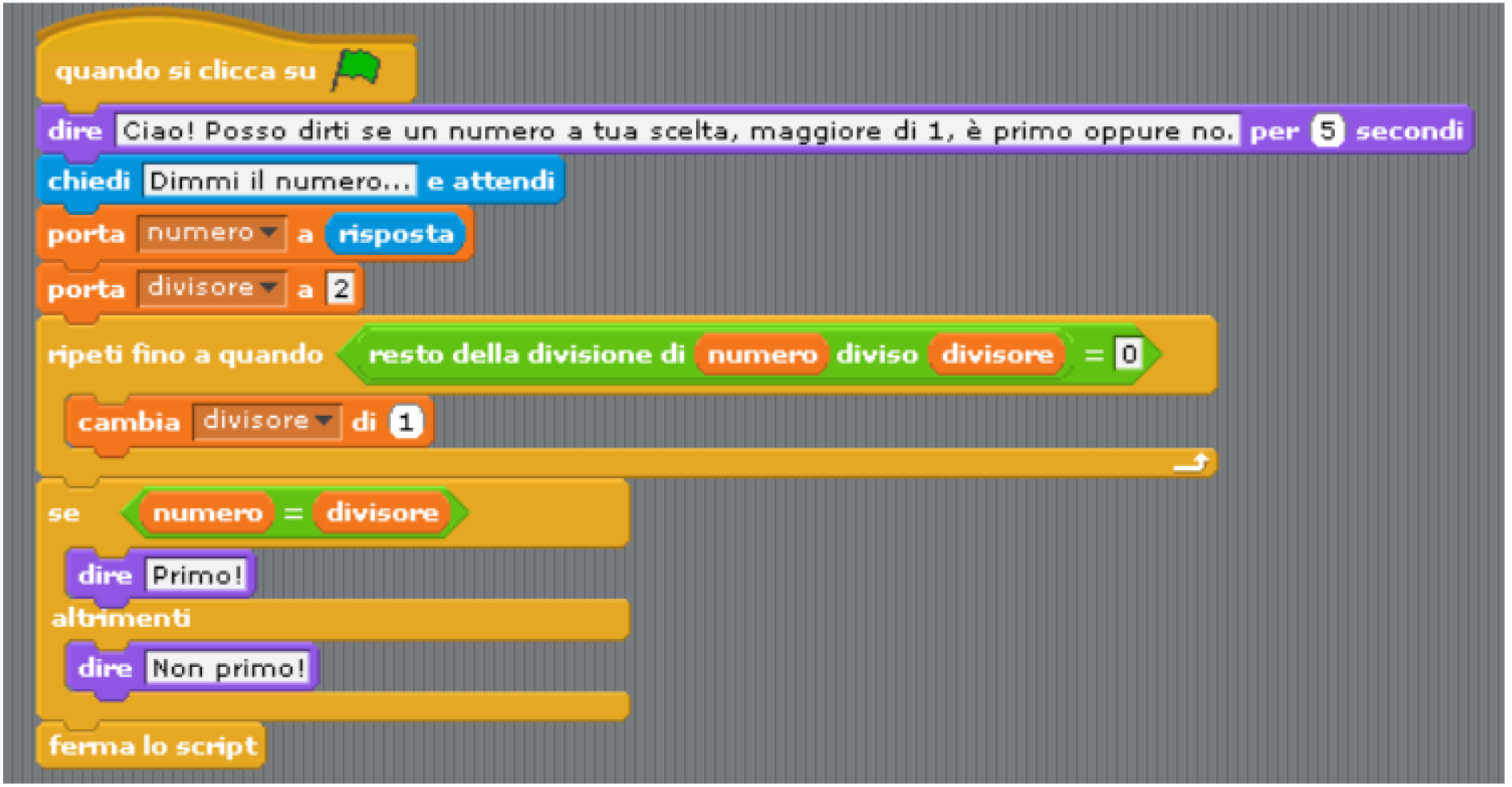
Per disegnare una scacchiera 8x8:

```
setpc white fd 300 lt 90 fd 400 rt 90
repeat 4 [
  repeat 4 [ quadratopieno 100 yellow  quadratopieno 100 green ]
  setpc white  bk 100 lt 90 fd 800 rt 90
  repeat 4 [ quadratopieno 100 green  quadratopieno 100 yellow ]
  setpc white  bk 100 lt 90 fd 800 rt 90
]
```

Alcuni esempi con Scratch https://scratch.mit.edu/


quando si clicca su
dire Ciao! Posso calcolare quoziente e resto di una divisione, per sottrazioni successive. per 5 secondi
chiedi Dimmi il dividendo... e attendi
porta dividendo a risposta
chiedi Dimmi il divisore... e attendi
porta divisore a risposta
porta quoziente a 0
porta resto a dividendo
ripeti fino a quando resto < divisore
porta quoziente a quoziente + 1
porta resto a resto - divisore
dire unione di unione di quoziente = e quoziente e unione di e resto = e resto
ferma lo script


quando si clicca su
dire Ciao! Posso dirti la data della Pasqua di un anno a tua scelta, dal 1900 al 2099. per 7 secondi
chiedi Dimmi l'anno! e attendi
porta anno a risposta
porta a a resto della divisione di anno diviso 19
porta b a resto della divisione di anno diviso 4
porta c a resto della divisione di anno diviso 7
porta d a resto della divisione di 19 * a + 24 diviso 30
porta e a resto della divisione di 2 * b + 4 * c + 6 * d + 5 diviso 7
se d = 28 e e = 6
dire 18 aprile!
altrimenti
se d = 29 e e = 6
dire 19 aprile!
altrimenti
porta f a d + e
se f > 9
dire unione di f - 9 e aprile!
altrimenti
dire unione di f + 22 e marzo!
ferma lo script



Estensione: Snap! https://snap.berkeley.edu/

Certi problemini sembrano facili …

Alcuni animali giungono in riva al fiume, uno dopo l'altro, per abbeverarsi.
Quando un animale arriva, manda via tutti quelli più piccoli di lui.
Data la sequenza delle taglie degli esemplari che giungono al fiume, secondo l'ordine di arrivo, quanti ne rimangono alla fine?

Soluzione più efficiente (vedi programmi_Python_2.7.9 > animali.py):

```
A = [3, 9, 2, 6, 4, 5, 2, 2]

quanti = len(A)
rimasti = 1
massimo = A[quanti-1]
# l'ultimo arrivato rimane e la sua taglia vale massimo
for i in range(quanti-2, -1, -1):
    # dal penultimo arrivato (quanti-2) sino a scendere al primo (0)
    if A[i] >= massimo:
        massimo = A[i]
        rimasti += 1
print "animali rimasti =", rimasti

>>>  animali rimasti = 5
```

- Data una sequenza di interi, anche negativi, calcolare la somma della sottosequenza di somma massima; due versioni con diverse complessità, una “quadratica” e una “lineare”: vedi programmi_Python_2.7.9 > somma_max.py

- Data una sequenza di bit, che rappresenta una partitura per tamburello (1 = batti un colpo, 0 = pausa), stabilire se è periodica e, in caso affermativo, trovare il periodo: vedi programmi_Python_2.7.9 > tamburello.py

- Data una sequenza di N bit (una strada con N lampioni, 1 = acceso, 0 = spento), si vuol sapere quanti lampioni bisogna accendere affinché ogni sottosequenza di M lampioni ne abbia almeno K accesi ($1 \leq K \leq M \leq N$); due versioni con diverse complessità: vedi programmi_Python_2.7.9 > lampioni_1.py e lampioni_2.py

Qualche problema un po' più difficile …

- Ogni persona in un insieme di N dà la propria disponibilità per un intervallo di giorni [da, a] compresi. Devono essere coperti da almeno una persona tutti i giorni da 0 a K – 1, impiegando il minor numero di persone: vedi programmi_Python_2.7.9 > turni.py

- Data una sequenza di N interi (una fila di luci di diversi colori, codificati con i numeri 0, 1, …, C – 1), calcolare la lunghezza della sottosequenza più corta che contiene almeno una luce di ciascun colore: vedi codifica in C++ nella cartella programmi_Cpp > luci_di_Natale

- Quanti sono gli alberi binari con N nodi, radice compresa, che hanno più nodi nel sottoalbero sinistro della radice rispetto al destro? Vedi codifica in C++ nella cartella programmi_Cpp > alberi_LR

Alcuni programmi nell'ambiente Maple

Nella cartella programmi_Maple sono contenuti quattro file commentati (sia i *worksheet*, sia i file in pdf con i risultati):

1) calcolo di: resto, mcd, fattoriale e numeri di Fibonacci, in forma sia ricorsiva sia iterativa; calcolo del mcm;
2) i numeri di Bernoulli;
3) i metodi di bisezione e di Erone per calcolare la radice quadrata; la serie di Gregory-Leibniz per approssimare π;
4) il triangolo di Tartaglia; i numeri di Mersenne e il problema della fattorizzazione.

Il problema della fattorizzazione (in forma decisionale) è uno dei pochi noti che appartengono a NP, ma non si sa se siano NP-completi o se appartengano (anche) a P.

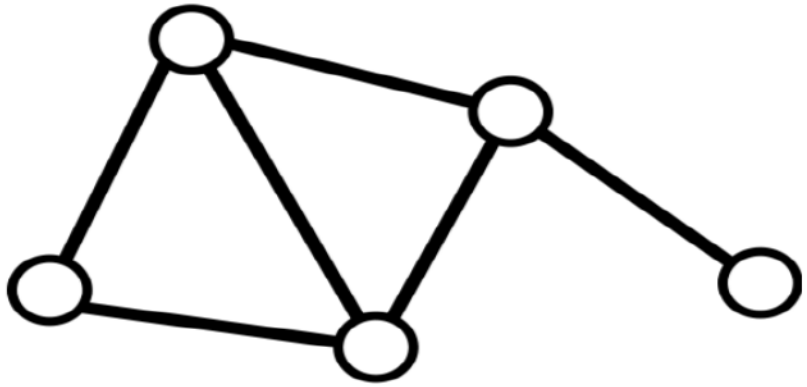
Un altro è il problema dell'isomorfismo tra grafi …

L'amico sconosciuto

(Slovenia, 2014)

Aldo dice che è amico di Bea, Clo e Davi,
Clo dice che è amica di Aldo ed Egle,
Egle è amica di Clo,
Davi si proclama amico di Aldo e Bea,
e infine Bea è amica di Aldo e Davi.

I cinque decidono di disegnare un diagramma di amicizia in cui rappresentano se stessi con un cerchietto e il rapporto di amicizia con un tratto.

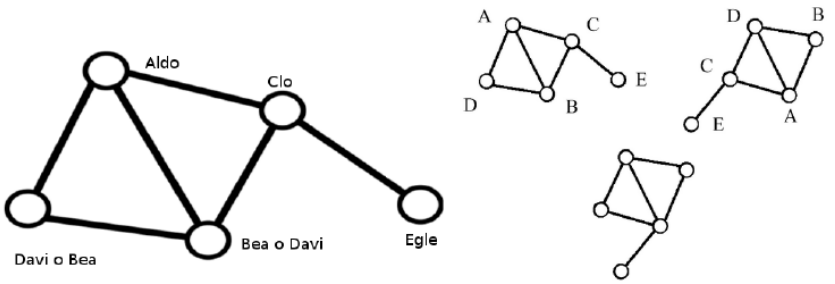


Ma hanno dimenticato di segnare i nomi.

Confrontando le amicizie note con il diagramma si scopre che c'è un'amicizia non menzionata. Che cosa si può dire in merito con certezza?

- ◯ Clo e Davi sono amici.
- ◯ Clo ha un altro amico o amica, ma non sappiamo chi sia.
- ◯ Egle ha un altro amico o amica, ma non sappiamo chi sia.
- ◯ Nessuna delle affermazioni precedenti è vera con certezza.

Soluzione.
La risposta corretta è la seconda dall'alto.



Approfondimenti. Due sono le assegnazioni dei nomi ai nodi che rispettano la relazione di amicizia data; ovviamente, i due grafi che ne risultano sono *isomorfi,* "hanno la stessa forma". Piú precisamente, due grafi sono isomorfi se risultano identici, eventualmente dopo aver rinominato in modo opportuno i nodi di uno dei due (sicché le loro rappresentazioni grafiche saranno equivalenti dal punto di vista topologico); ad esempio, i due grafi piccoli in alto a destra sono isomorfi, ma nessuno dei due è isomorfo col terzo in basso (indipendentemente dai nomi attribuibili ai suoi nodi). Per inciso, il problema dell'*isomorfismo tra grafi* (dati due grafi arbitrari, stabilire se sono isomorfi) è uno dei pochi conosciuti che appartengono alla *classe NP* (la classe dei problemi decisionali per i quali non è noto alcun algoritmo efficiente che li *risolva*, né si sa se esista, tuttavia è noto un algoritmo efficiente che *verifichi* le risposte affermative), ma per i quali non è stato dimostrato né che siano *NP-completi* (i problemi "piú difficili" della classe NP: qualsiasi problema in NP può essere "trasformato", in modo efficiente, in uno di essi) né che appartengano alla *classe P* (i problemi per i quali è noto un algoritmo risolutivo efficiente). Con "efficiente" s'intende in tempo di ordine al piú polinomiale rispetto alla dimensione del problema (o, per essere piú precisi, rispetto alla lunghezza in bit dei dati di input).

Parole chiave: Grafi, isomorfismo fra grafi, problemi NP.

Ancora nell'ambiente Maple …

Nella cartella programmi_Maple trovate anche il file (in pdf) Introduzione ai sistemi dinamici non lineari, che sviluppai oltre un decennio fa, nell'ambito del Progetto Problem Posing & Solving; un tema che potrebbe essere trattato in un corso sia di Fisica sia di Matematica, corredato di numerosi problemi risolti con Maple, suddiviso in capitoli:

- un semplice sistema caotico
- il modello differenziale
- i sistemi di Lotka-Volterra
- il sistema prede-predatori
- oscillatori
- il circuito di van der Pol
- biforcazioni di Hopf

Un problema dato alla finale Coppa Student Kangourou 2023

Quesito 7 – Piccoli roditori crescono

Studiando l'evoluzione di una popolazione di piccoli roditori in una certa regione, gli scienziati hanno formulato un modello che spiega l'andamento nel tempo della quantità di esemplari, tenendo conto dell'arrivo di una specie predatrice che ne limita la crescita. Detta r(t) la quantità di roditori (in migliaia), funzione del tempo t (in anni), essa è la soluzione dell'equazione differenziale (non lineare)

$$dr(t)/dt = r(t)/4 - r(t)^2/12$$

con r(0) = 1. Applicando il più semplice metodo di integrazione numerica (di Eulero, a un passo, esplicito), con passo h = 0.2, calcolate r(5) e date come risposta le prime due cifre significative del risultato ottenuto.
Nota: il metodo citato approssima y(x), soluzione dell'equazione dy/dx = f(x, y), con la successione y(i+1) = y(i) + h·f(x(i), y(i)), dato y(0) e fissato un valore opportuno per il passo h.

(Il quesito è risolto da programmi_Python_2.7.9 > roditori.py riportato anche nella slide seguente.)

La risposta può essere trovata eseguendo questo programma in Python:

```
def f(x, y):
    return y/4*(1 - y/3)
    # qui f non dipende da x

h = 0.2
x = 0.0
y = 1.0
for i in range(1, 26):
    y += h*f(x, y)
    x += h
    print x, y
```

La soluzione esatta dell'equazione differenziale data è la famiglia di funzioni

$r(t) = 3 / (1 + c \cdot \exp(-t/4))$

con c reale (provatelo per esercizio); dalla condizione iniziale $r(0) = 1$, si determina $c = 2$.
Ciascuna funzione di tale famiglia tende a 3 per t tendente a $+\infty$ (come si può verificare anche in base all'equazione data, studiando i punti di equilibrio); dunque $r(5) \approx 1.907$.
Nota: nell'equazione data, 4 ha come dimensione l'anno, 12 le migliaia di esemplari × anno.

Modelli più accurati devono considerare almeno l'interazione tra le due specie, le prede e i predatori, nel loro habitat naturale; il più semplice di essi fu proposto nel 1926 dal matematico Vito Volterra… (si veda la mia "introduzione ai sistemi dinamici non lineari").

Un altro problema della finale Coppa Student Kangourou 2023

Quesito 12 – La crescita della popolazione mondiale

Nel 1995 il fisico russo Sergei P. Kapitsa propose un modello di crescita nel tempo della popolazione mondiale p(t) assai aderente a quanto in effetti è avvenuto nella storia dell'umanità, e in particolare alla crescita demografica degli ultimi secoli. Ecco il modello:

$$dp(t)/dt = C / ((T - t)^2 + \tau^2)$$

dove il tempo t è espresso in anni (qui, con "anni", si intende anni della nostra era, ad esempio 2023), C = 186 (miliardi di persone × anni), T = 2007 (anni) e τ = 42 (anni).
Per calcolare numericamente la primitiva p(t) che ci interessa, partiamo dal dato relativo all'anno 1995, in cui il modello prevedeva una popolazione di circa 5.724 miliardi di persone, di poco inferiore alla stima reale. Poniamo quindi t = 1995.0 e p = 5.724 (miliardi di persone alla fine di quell'anno), e applichiamo la nota regola di integrazione detta "dei trapezi", procedendo col passo di un anno, ovvero sommando a p la quantità:

$$(g(t) + g(t + 1.0)) / 2.0$$

dove g = dp/dt, e incrementando poi t di 1.0 (anni), in modo da ottenere in p un'approssimazione della popolazione alla fine dell'anno t.
Iterando questo calcolo, alla fine di quale anno la popolazione mondiale ha superato gli 8 miliardi?

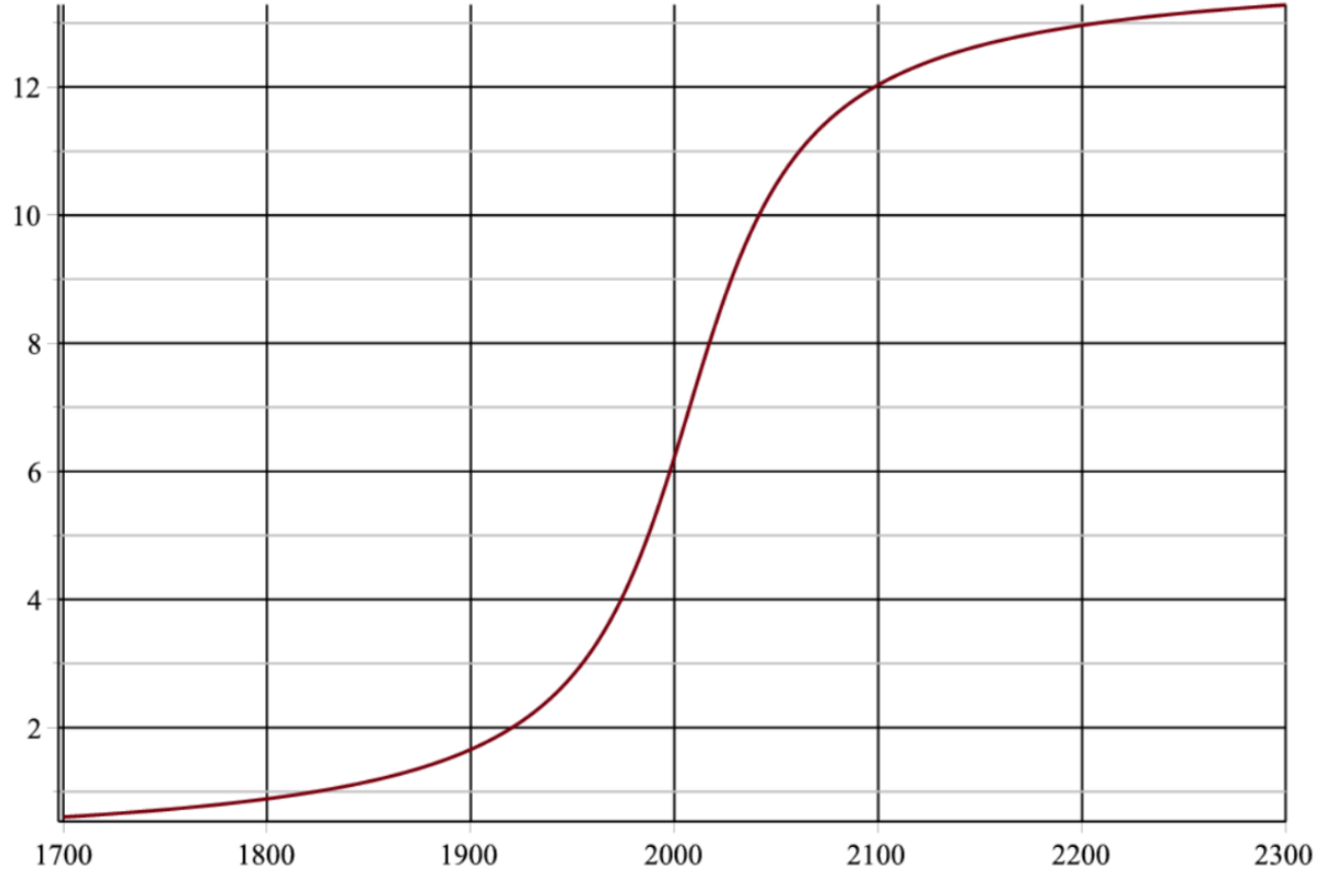
(Il quesito è risolto da programmi_Python_2.7.9 > popolazione.py riportato anche nella slide seguente.)

La risposta giusta, 2018, può essere trovata eseguendo questo programma in Python, dove non ci siamo premurati di minimizzare le valutazioni della funzione integranda:

```
def g(t):
    return 186 / ((2007 - t)**2 + 1764)

h = 1.0
t = 1995.0
p = 5.724
for i in range(1, 25):
    p += (g(t) + g(t+h))*h/2
    t += h
    print t, p
```

In realtà, è stato stimato che il raggiungimento degli 8 miliardi sia avvenuto verso la fine dello scorso anno 2022.

La primitiva esatta che ci interessa è $p(t) = (C/\tau)\cdot\text{arccot}((T - t)/\tau)$ (verificatelo per esercizio); la riportiamo nel grafico qui sotto, con i parametri sopra fissati, per t che va dall'anno 1700 (dove vale poco più di 600 milioni) all'anno 2300. La soglia degli 8 miliardi è superata nel 2018 (dove questa funzione vale circa 8.09077), in accordo con la nostra simulazione numerica.



Secondo questo modello, la popolazione mondiale supererà i 10 miliardi nel 2042, e nei secoli successivi tenderà a stabilizzarsi a poco più di 13,9 miliardi (ricordiamo che arccot(x) tende a π per x tendente a $-\infty$).

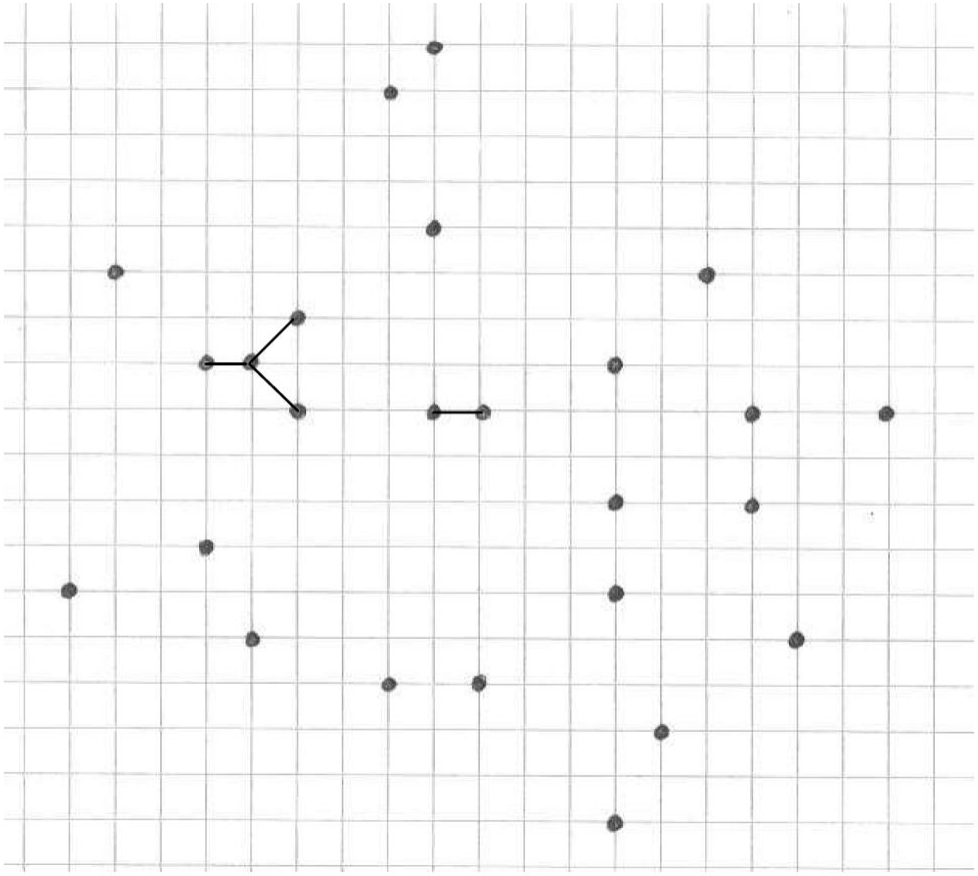
Attualmente, con i dati degli ultimi decenni, alcune fonti prevedono il traguardo dei 10 miliardi nel 2050, altre una punta di 9.7 miliardi verso il 2065 seguita poi da un calo a 8.8 miliardi alla fine di questo secolo XXI: chi avrà ragione? Non ci resta che attendere!

Partendo dallo scenario di un compito da risolvere (ovvero da una particolare istanza del problema):

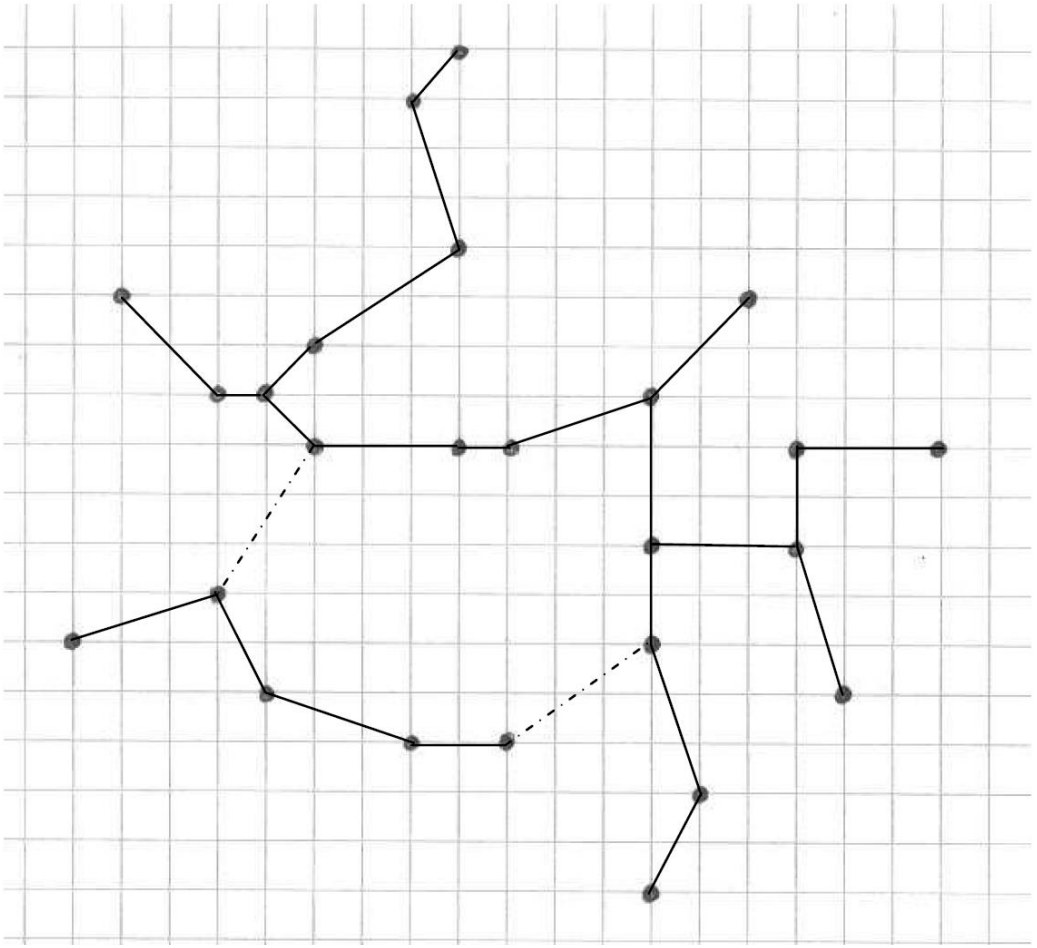
- ❑ *Algoritmi "efficienti" su grafi (pesati / non pesati)*
  - *Minimum Spanning Tree* (Borůvka, Kruskal, Prim-Jarník → Dijkstra per cammini minimi)
  - Visite in profondità e in ampiezza …
- ❑ *Problemi "difficili" su grafi (pesati / non pesati)*
  - *Travelling-Salesman Problem* (di permutazione)
  - *Min. Vertex Cover* e *Max. Independent Vertex Set* (due problemi di sottoinsieme, complementari) …
- ❑ *Altri problemi "difficili"*
  - *Rectilinear Steiner Tree*, *Bin-Packing* (di partizione), *Knapsack*, *Multiprocessor Scheduling* …

Gli elettricisti – 4 punti

Due elettricisti, Crusca e Primo, devono collegare con cavi elettrici 25 *punti luce*, disposti come illustrato nella figura qui sotto, in modo tale che da ciascun punto siano raggiungibili tutti gli altri, ma senza che si formino anelli (ossia percorsi chiusi), e impiegando la minor lunghezza possibile di cavo elettrico. L'apprendista Primo pensa che sia un compito difficilissimo. L'esperto Crusca, però, gli spiega che basta collegare via via la coppia di punti più vicini, come si è già cominciato a fare nella figura, evitando solo di formare anelli. Completate i collegamenti seguendo il consiglio di Crusca.



Soluzione



La soluzione non è unica. I due segmenti tratteggiati, di stessa lunghezza, chiudono un anello: bisogna dunque scegliere *uno solo* dei due per completare la rete elettrica.

L'obiettivo consiste nel collegare i punti luce in modo che nessuna porzione di rete rimanga isolata dal resto, senza creare anelli: in altre parole, considerati due punti luce qualsiasi, il collegamento tra loro, costituito da uno o più segmenti di filo, deve essere *unico*! Inoltre, la somma delle lunghezze di tutti i segmenti di filo utilizzati deve essere la minore possibile.

Se in ogni punto luce immaginiamo sia collocata una lampadina, collegando allora due punti luce qualsiasi ai morsetti di una batteria si illuminerà quell'unico percorso che li unisce, che – si osservi – può essere costituito da più segmenti non allineati (e quindi *non* è in generale il più breve costruibile tra i due punti considerati), proprio perché ciò che si deve minimizzare è la lunghezza *totale* dei collegamenti.

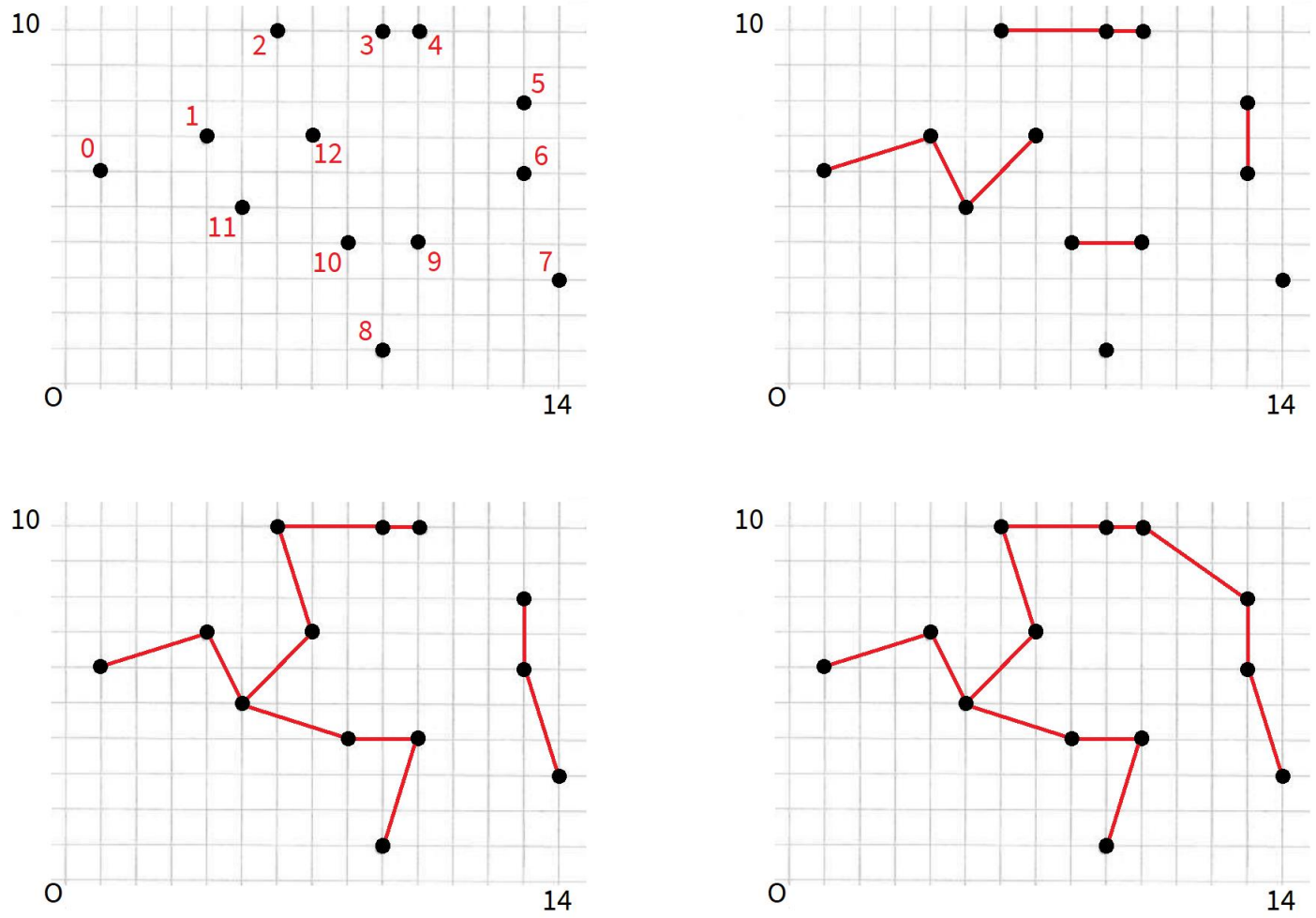
Una possibile difficoltà consisteva nel *valutare le lunghezze* dei collegamenti: se le misuriamo usando come unità il lato di ciascun quadretto, i due segmenti tratteggiati – ad esempio – hanno ciascuno lunghezza pari alla radice quadrata di $13 = 2^2 + 3^2$, per il *teorema di Pitagora*, applicato a un triangolo rettangolo di cateti 2 e 3 quadretti. Ma dovendo solo valutare se un lato è più lungo di un altro, non c'è alcun bisogno di calcolare radici: si possono semplicemente confrontare i quadrati. Così i due segmenti tratteggiati hanno sicuramente lunghezza minore di 4, perché $4^2 = 16 > 13$. Non era comunque vietato usare un righello...

Anche questa è informatica!

Il consiglio dell'esperto Crusca corrisponde in realtà al ben noto *algoritmo di Kruskal*, mentre un procedimento alternativo è noto come *algoritmo di Prim*, il che dovrebbe giustificare i nomi degli elettricisti.
Come funziona l'algoritmo di Prim? Si parte da un punto arbitrario e lo si collega al punto più vicino, poi ad ogni passo successivo si collega un nuovo punto: il più vicino a qualcuno di quelli già collegati!

→ … Naturalmente, bisogna dimostrare la correttezza di questi algoritmi!

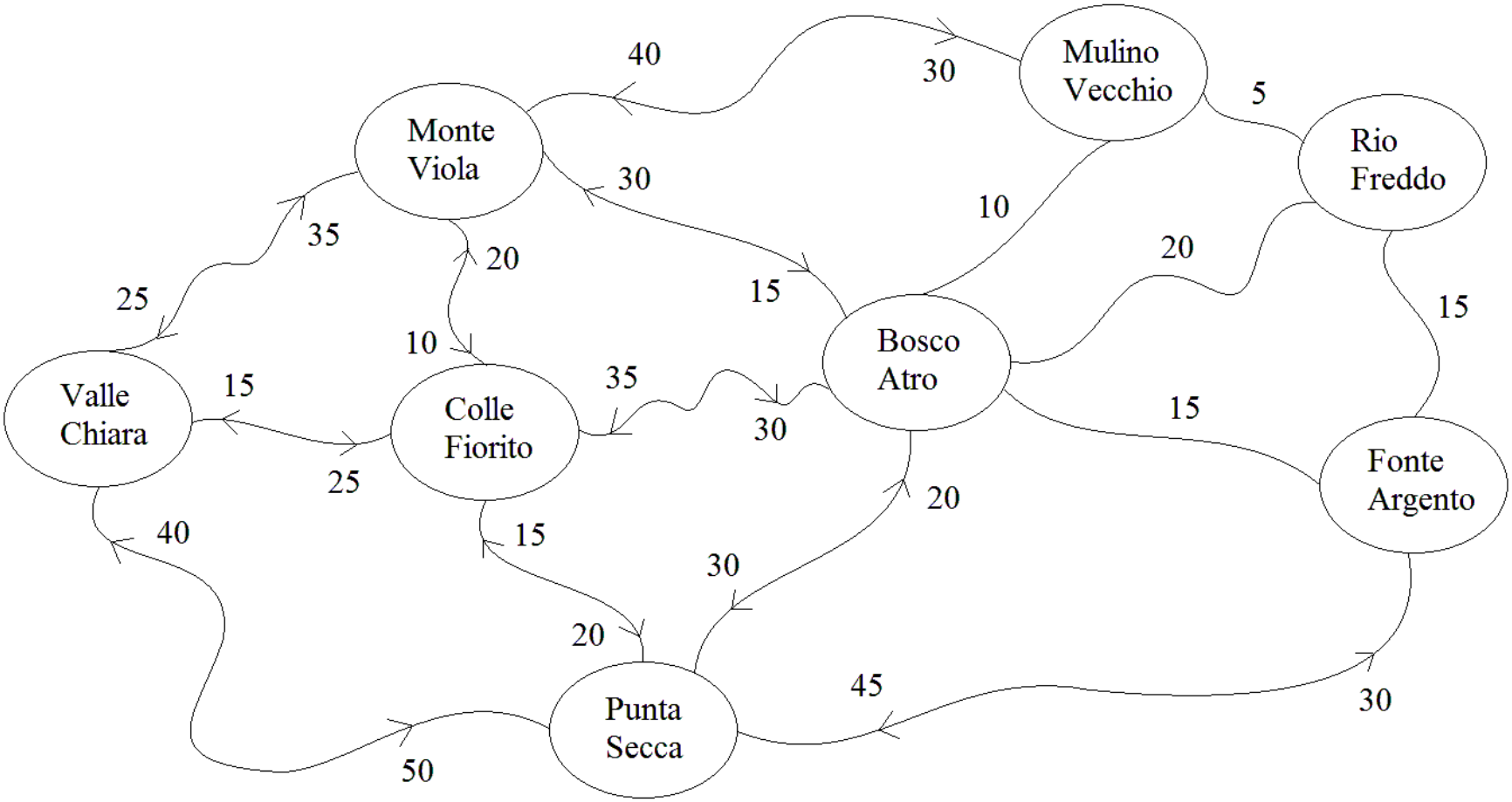
Algoritmo di Kruskal (1956)



Vedi codifica in C++ nella cartella programmi_Cpp > Kruskal Sembra facile…

ma l'idea giusta inizia col ripartire i punti da collegare in altrettanti insiemi…

Qual è il cammino più breve da Valle Chiara a Fonte Argento?

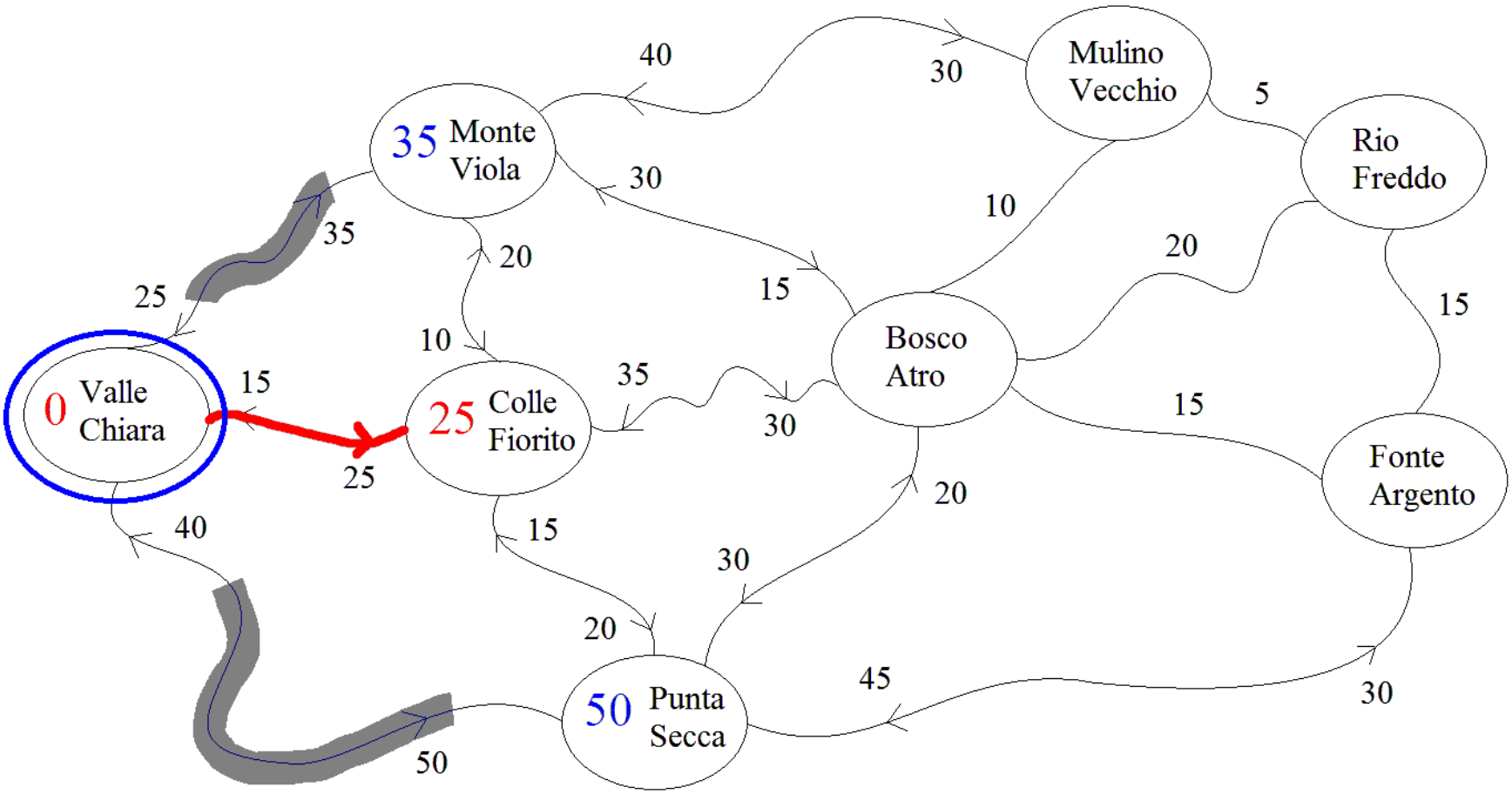


Algoritmo di Dijkstra (1959)

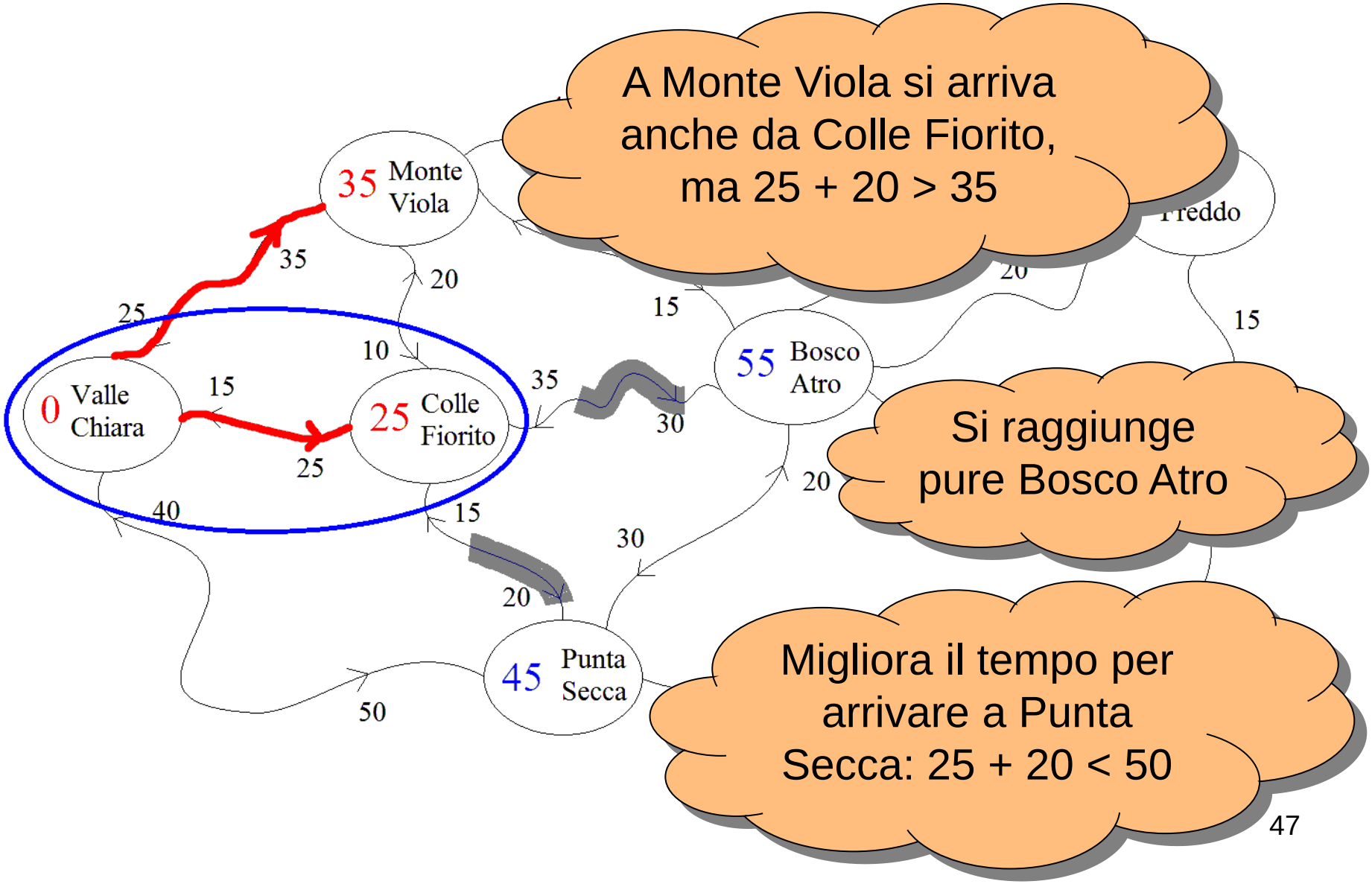
- la tecnica è "*greedy*": si costruisce un insieme S di nodi, iniziando col solo nodo di partenza v_0
- ad ogni passo, si aggiunge all'insieme S il nodo "più vicino": tra quelli a distanza di un arco da un nodo di S, considerando i costi per raggiungerli a partire da v_0 e toccando soltanto nodi di S
- dopo $n - 1$ passi, l'insieme S è costituito da tutti gli n nodi, e sono noti i cammini di costo minimo da v_0 a ciascuno degli altri nodi (raggiungibili)

Vedi codifica in C++ nella cartella programmi_Cpp > Dijkstra

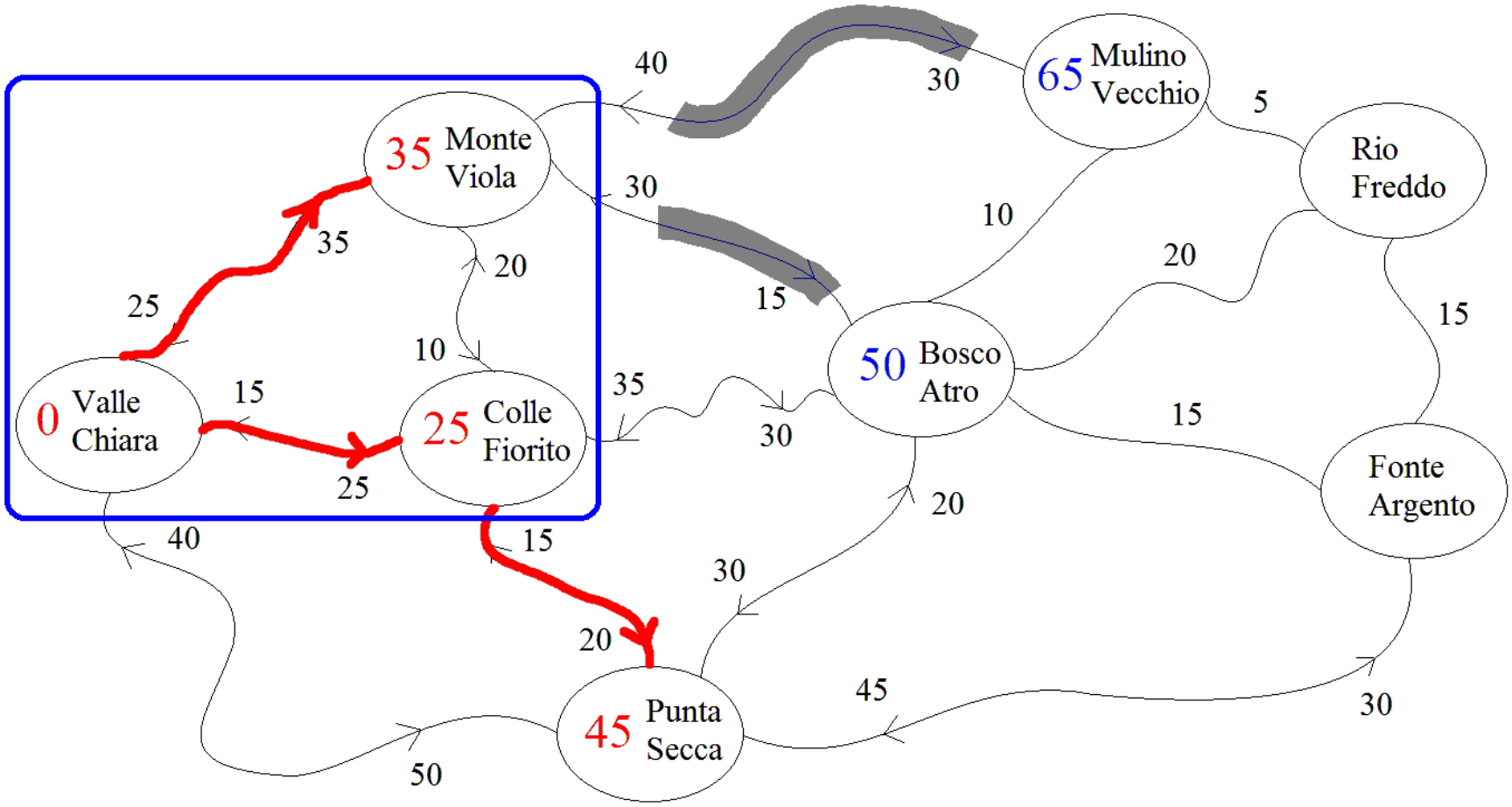
Algoritmo di Dijkstra primo passo



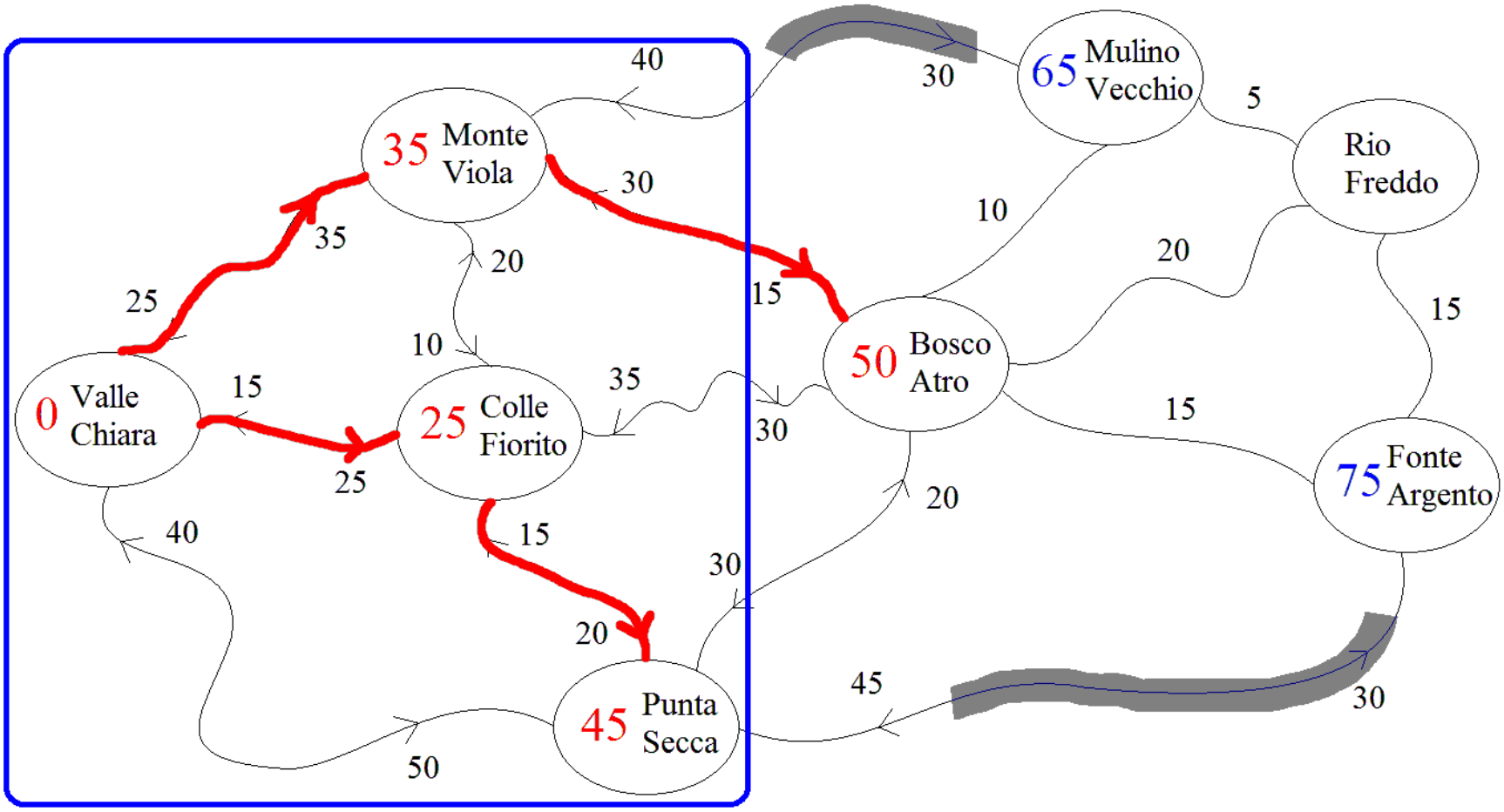
Algoritmo di Dijkstra — secondo passo



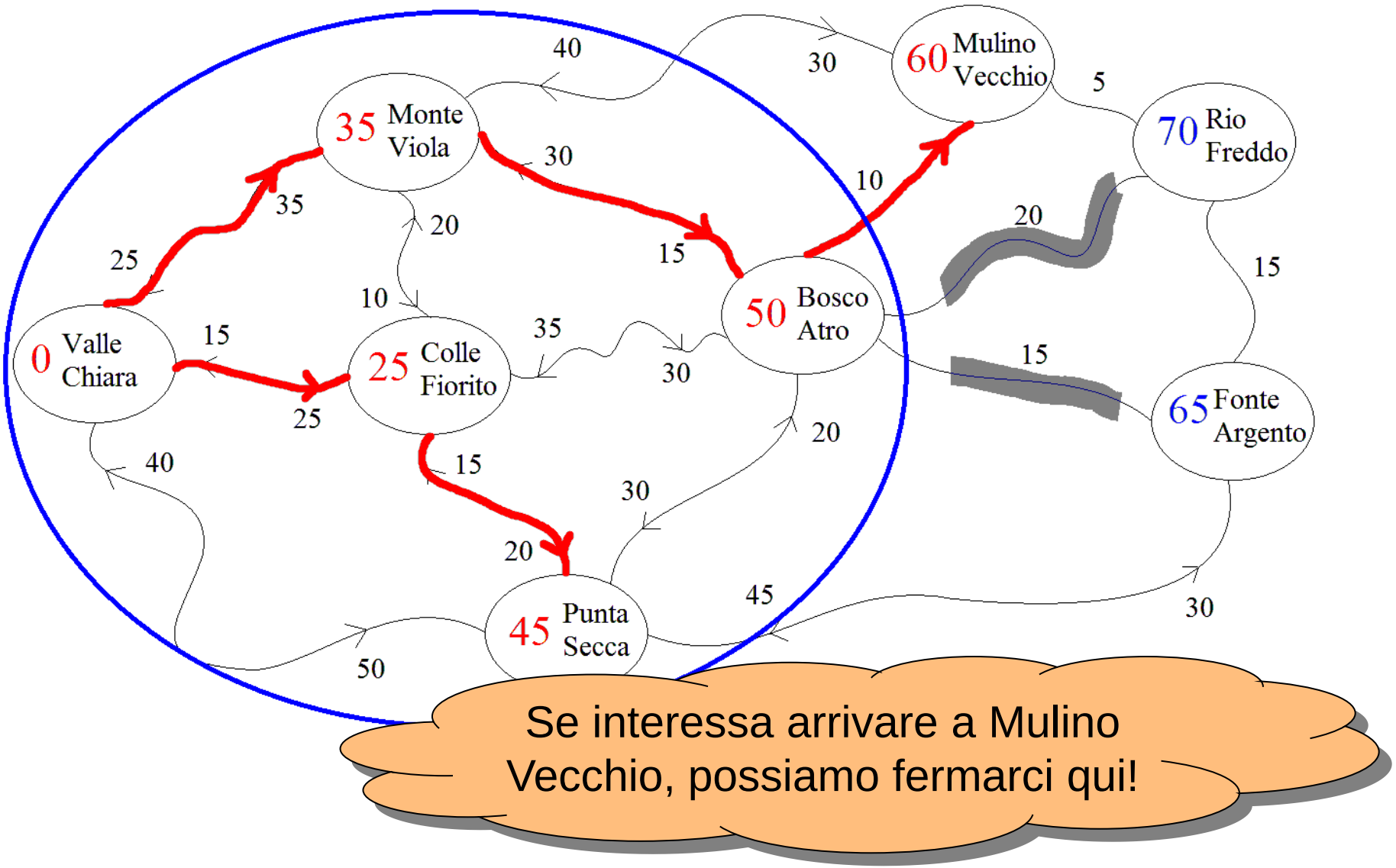
Algoritmo di Dijkstra terzo passo



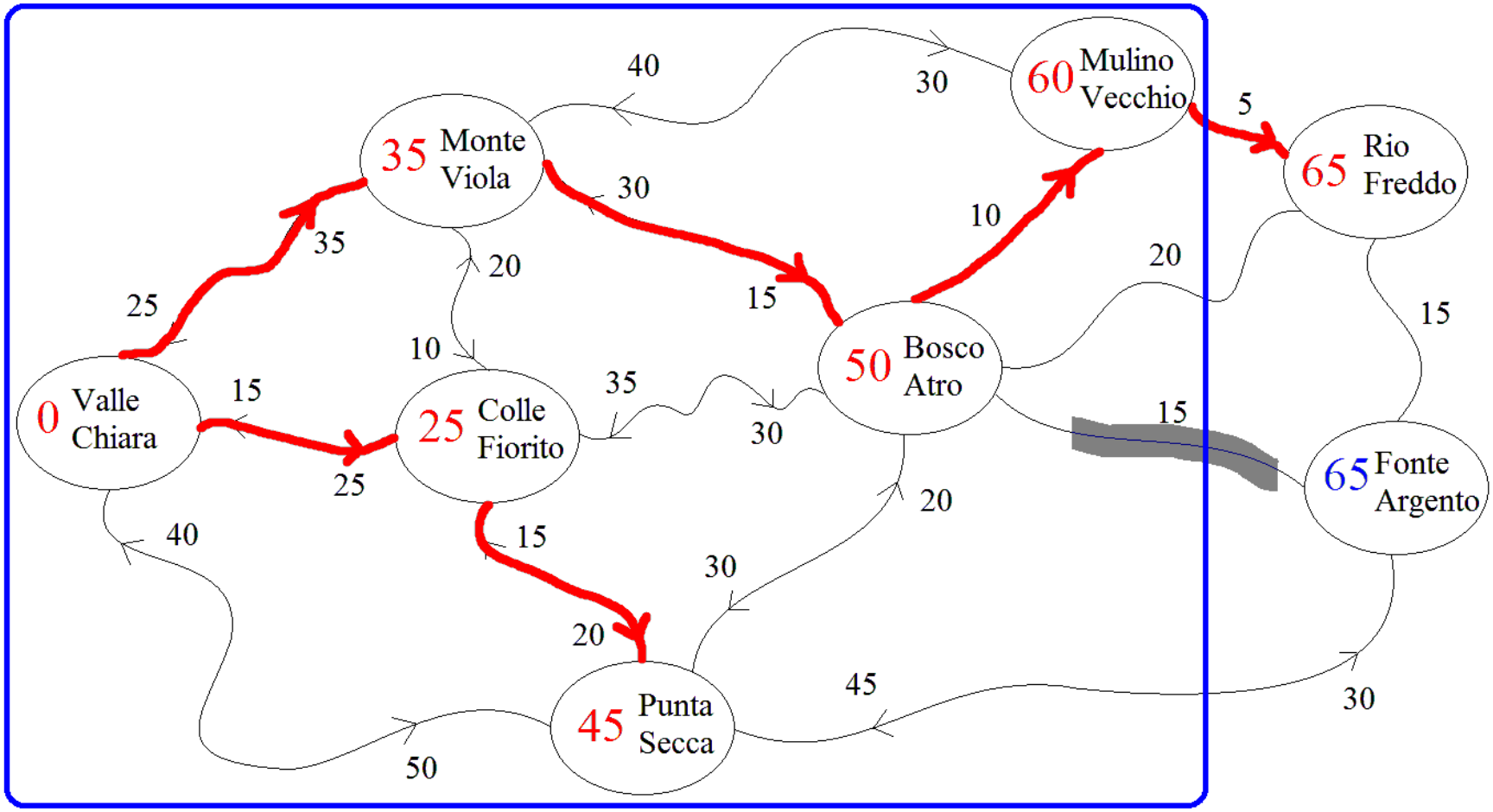
Algoritmo di Dijkstra quarto passo



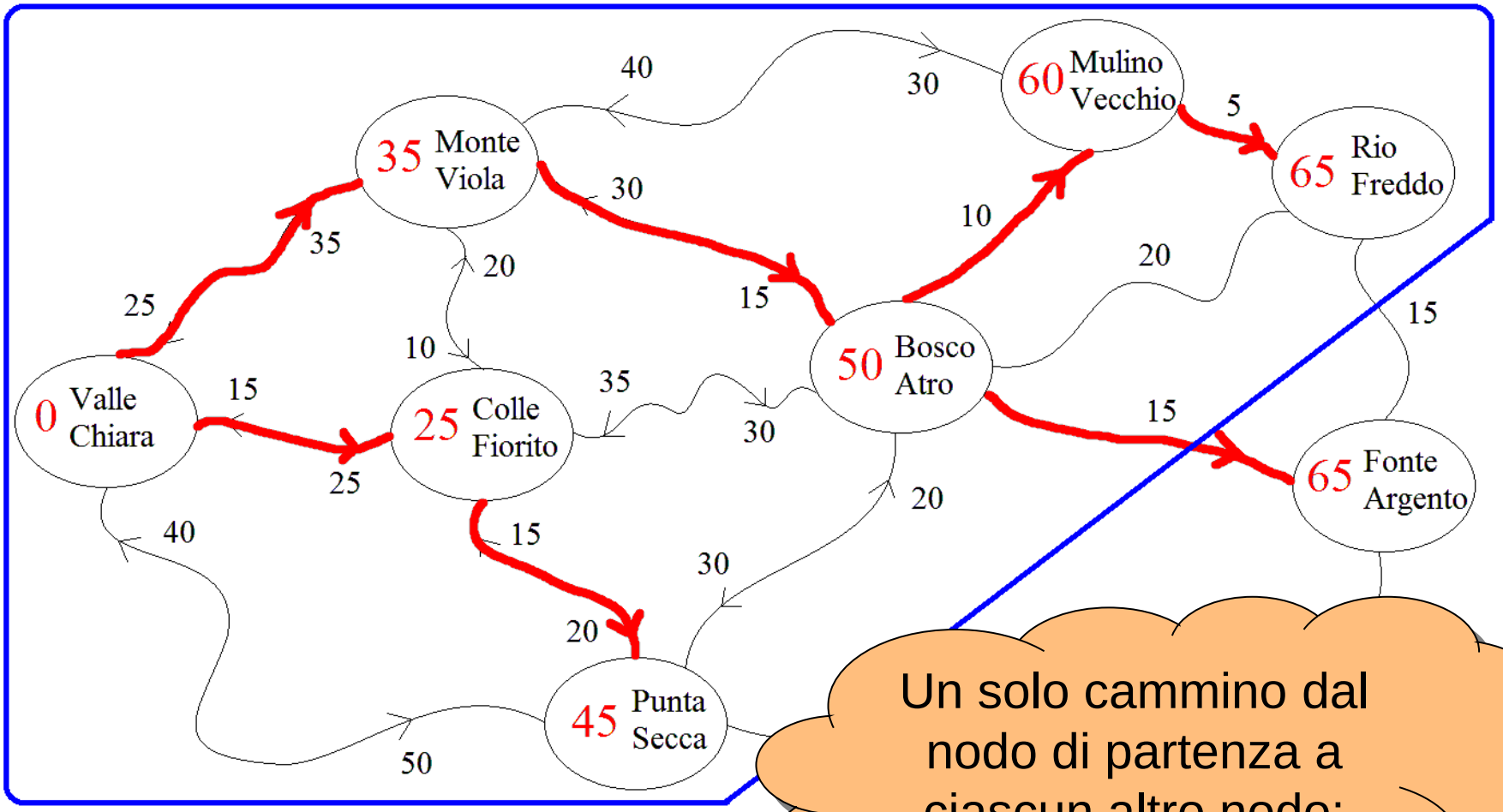
Algoritmo di Dijkstra quinto passo



Algoritmo di Dijkstra sesto passo



Algoritmo di Dijkstra — settimo passo



Algoritmo di Dijkstra: “complessità”

- nel caso peggiore, al crescere del numero n dei nodi, il tempo di elaborazione tende ad aumentare in modo proporzionale a n^2

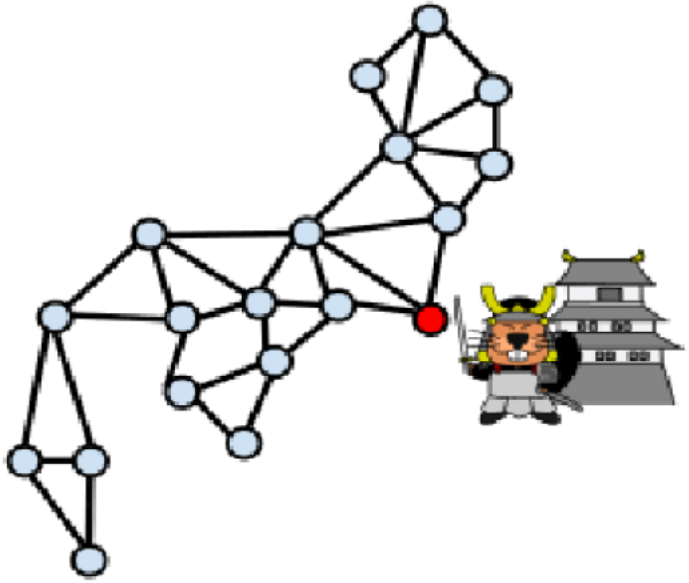
- si può migliorare utilizzando strutture di dati opportune; comunque è efficiente (e quindi il problema è trattabile)

- trovare il cammino “più lungo” tra due nodi, o un cammino che tocchi una e una sola volta ciascun nodo: per questi problemi non si conoscono (né si sa se esistano) algoritmi efficienti in generale!

Segnali di fumo (2 punti)

(Giappone, 2014)

Lo storico greco Polibio (206-124 a.C.) descrisse un sistema di comunicazione basato su torri di segnalazione. Anche in Giappone durante il periodo degli shogunati fu realizzato un sistema di torri per comunicare con segnali di fumo in caso di emergenza. Nella figura il punto rosso indica la sede dello shogunato.
I punti blu indicano le torri di segnalazione e due punti sono uniti da un tratto se le due torri sono reciprocamente visibili. I responsabili di ogni torre accendono i loro fuochi esattamente un minuto dopo che hanno avvistato un segnale proveniente da un'altra torre.

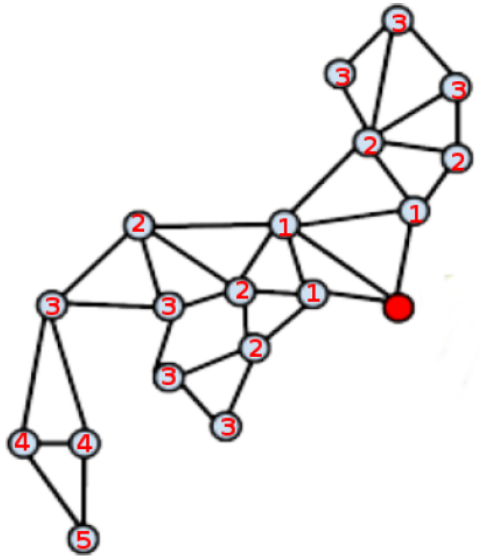


Quanto tempo ci vuole perché una segnalazione in partenza dalla sede dello shogunato abbia senz'altro raggiunto anche le più remote torri dell'impero e tutti i fuochi siano accesi?

- 8 minuti
- 5 minuti
- 4 minuti
- 6 minuti

Visita in ampiezza

Soluzione.
Per raggiungere anche le piú remote torri dell'impero servono 5 minuti, come si vede nella figura, nella quale sono stati indicati i minuti necessari per raggiungere ogni torre.



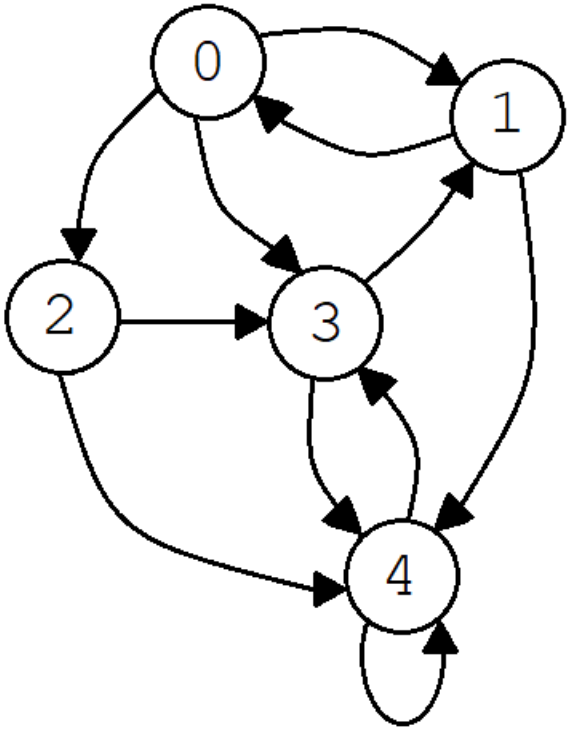
Anche questa è informatica! Il quesito proposto è un'istanza del problema di trovare un cammino di costo minimo da un nodo (dato) a ciascuno degli altri nodi in un grafo (non orientato e connesso), dopodiché si potrà conoscere il maggiore di tali costi. Per risolvere questo problema in generale esistono diversi algoritmi efficienti.

Nel nostro caso particolare, tutti gli archi hanno il medesimo "costo" (il tempo di un minuto) e, in virtú di questo fatto, è sufficiente una *visita in ampiezza*: costruiamo un insieme di nodi, mettendovi inizialmente soltanto quello di partenza (che rappresenta il punto rosso); al passo t ($t = 1, 2, \ldots$) aggiungiamo all'insieme tutti i nodi (che ancora non vi sono inclusi) collegati da un arco a qualcuno dei nodi che già vi sono inclusi: i nodi aggiunti sono precisamente quelli che "distano" t minuti dal nodo di partenza. All'ultimo passo saranno inclusi i nodi piú distanti, e il valore attuale di t ci dirà quanti minuti essi distano dal nodo di partenza.

Parole chiave e riferimenti: Grafo, cammino minimo, visita in ampiezza.

Backtracking: tutti i percorsi da-a in un di-grafo

```
5  11  0  4
0  1
0  2
0  3
1  0
1  4
2  3
2  4
3  1
3  4
4  3
4  4
```



6 percorsi aciclici da 0 a 4

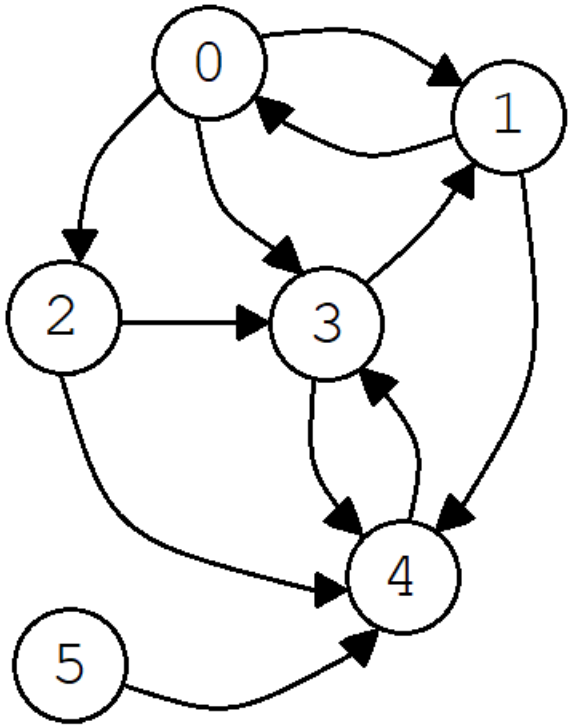
```
0  1  4
0  2  3  1  4
0  2  3  4
0  2  4
0  3  1  4
0  3  4
```

Vedi codifica in C++, basata sulla visita in profondità

nella cartella programmi_Cpp > percorsi_alternativi

Chiusura transitiva: un problema più semplice

```
6  11
0  1
0  2
0  3
1  0
1  4
2  3
2  4
3  1
3  4
4  3
5  4
```



chiusura transitiva

```
1  1  1  1  1  0
1  1  1  1  1  0
1  1  1  1  1  0
1  1  1  1  1  0
1  1  1  1  1  0
1  1  1  1  1  0
```

aciclico: falso

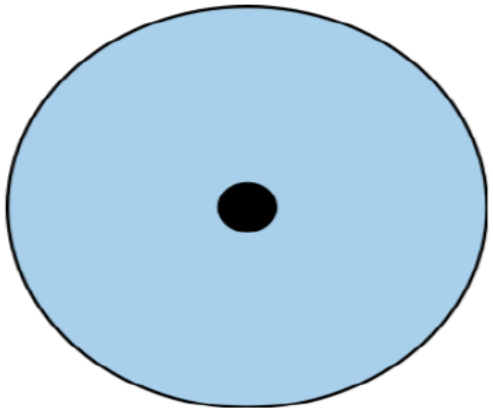
Vedi codifica in C++ dell'algoritmo di Warshall (1962)

nella cartella programmi_Cpp > chiusura_transitiva

Network

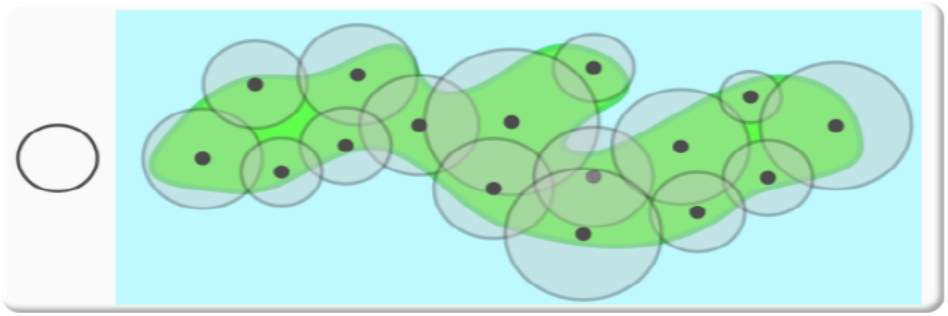
(Ungheria, 2014)

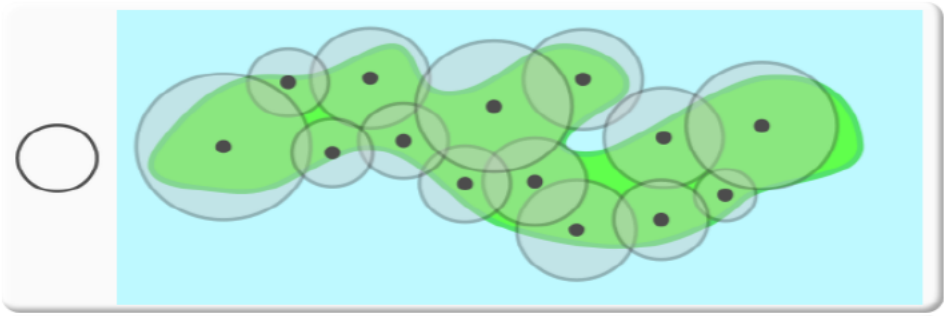
La compagnia di telecomunicazioni Grancastoro deve installare antenne per cellulari sull'Isola dei Castori; vuole costruire una rete molto affidabile: se un'antenna si guasta, le altre devono poter continuare a comunicare tra loro.

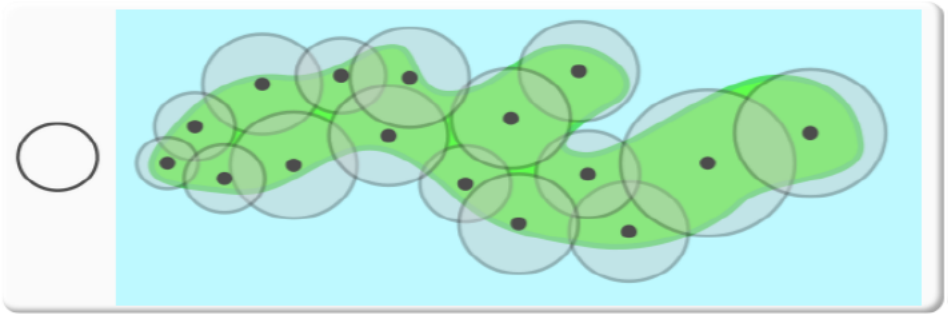


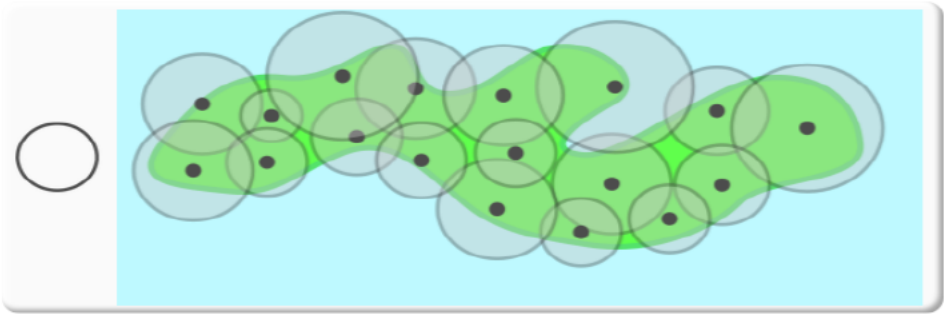
La figura mostra un'antenna e la sua area di copertura. Quando due aree si sovrappongono, anche di poco, le due antenne possono comunicare.

Quale fra le seguenti disposizioni funziona anche se si guasta una (ma non più di una) antenna qualsiasi?

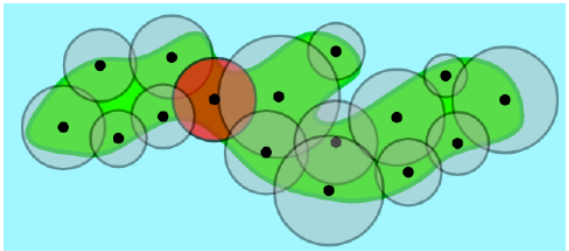


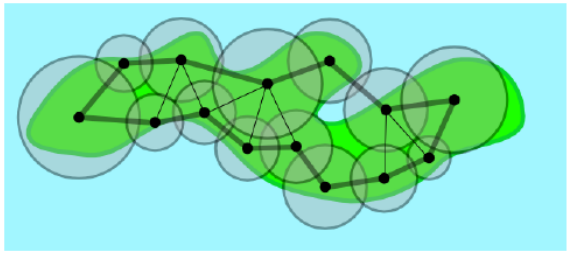


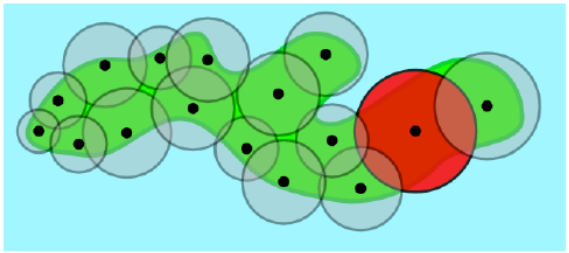


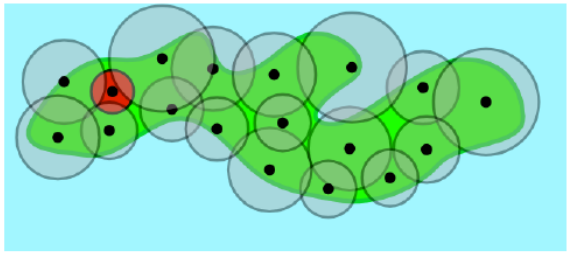


Suggerimento: poiché il grafo è non orientato, basta costruire un albero di visita a partire da un nodo a scelta e verificare se tutti i nodi sono stati raggiunti… Questo va fatto dopo aver tolto un arco ogni volta.









Soluzione.
La risposta corretta è quella in alto a destra.

Approfondimenti. Il problema può essere modellato per mezzo di un *grafo non orientato*: i *nodi* rappresentano le antenne e due nodi distinti sono collegati da un *arco* (non orientato) se e soltanto se le due antenne da essi rappresentate possono comunicare direttamente tra loro (cioè le rispettive aree di copertura si sovrappongono). Si tratta allora di stabilire se, partendo da un grafo *connesso* (cioè ove da ciascun nodo sono raggiungibili tutti gli altri) e togliendo uno degli archi, il grafo rimane connesso, qualunque sia l'arco eliminato.

Non resta che provare per ognuno degli archi: si elimina l'arco in questione e si effettua un *test di connessione* sul grafo cosí ridotto. Per fare questo test, si può scegliere *un* nodo a piacere, costruire un albero di visita (in ampiezza o in profondità) a partire da esso, e controllare infine se tutti i nodi sono stati raggiunti; rappresentando il grafo con la sua matrice di adiacenza (simmetrica), un test di connessione richiede un tempo di ordine quadratico rispetto al numero dei nodi.

Parole chiave: Grafo non orientato, grafo connesso.

Problemi (risolubili) “trattabili”

- Problemi risolubili in tempo polinomiale:

  esiste (ed è noto) *almeno un* algoritmo risolutivo che richiede tempo polinomiale nella lunghezza dell’input espressa in bit, e quindi possono essere risolti in modo efficiente (sempre entro certi limiti).

Esempi:

• ordinare una sequenza arbitraria di n numeri naturali: esistono vari algoritmi che richiedono un tempo d’esecuzione della forma $an^2 + bn + c$, ma anche algoritmi più efficienti …

• trovare il percorso più breve tra ciascuna coppia di nodi in un grafo qualsiasi; c’è anche un algoritmo assai compatto, che ha complessità cubica: Floyd, 1962, stessa idea di Warshall …
(vedi codifica in C++ nella cartella programmi_Cpp > Floyd)

Problemi (risolubili) di fatto “intrattabili”

- Problemi intrinsecamente esponenziali:

  qualsiasi algoritmo risolutivo (noto o non noto) richiede un tempo che dipende *almeno esponenzialmente* dalla lunghezza dell’input.

Esempi:

- (piuttosto banali) generare tutti gli anagrammi di una parola o elencare tutte le mosse per spostare una torre di Hanoi
- *decidere* se due espressioni regolari (con l’operatore quadrato) generano lo stesso linguaggio (A. R. Meyer e L. J. Stockmeyer, 1972)
- analizzare il gioco del blocco stradale o della dama $n \times n$.

Il confine tra queste due classi di problemi non è affatto netto: è una zona misteriosa, dove stanno tanti problemi interessanti …

Ladro chi ruba e chi riempie lo zaino (5 punti)

Un ladro sta svaligiando un negozio di elettrodomestici e naturalmente vuole accumulare refurtiva con piú valore possibile, ma ha uno zaino che regge al massimo 82 etti. Nel negozio ci sono oggetti di 4 tipi e per ciascun tipo ci sono 10 oggetti.

Tipo	Peso in etti	Valore in euro
Televisore	80	605
Cellulare	9	65
Microonde	90	100
Hard disk esterno	16	120

- Se la strategia usata dal ladro per riempire lo zaino è di scegliere sempre l'oggetto *piú leggero* che ci sta, qual è il valore della refurtiva che riuscirà a portare via?
- Se la strategia usata dal ladro per riempire lo zaino è di scegliere sempre l'oggetto *di maggior valore* che ci sta, qual è il valore della refurtiva che riuscirà a portare via?
- Se la strategia usata dal ladro per riempire lo zaino è di scegliere sempre l'oggetto *che ha il miglior rapporto valore/peso* che ci sta, qual è il valore della refurtiva che riuscirà a portare via?
- Per ottenere il massimo valore della refurtiva, sapete fare meglio?

Soluzione

Tipo	Peso in etti	Valore in euro
Televisore	80	605
Cellulare	9	65
Microonde	90	100
Hard disk esterno	16	120

- Se la strategia usata dal ladro per riempire lo zaino è di scegliere sempre l'oggetto *piú leggero* che ci sta, qual è il valore della refurtiva che riuscirà a portare via?

  9 cellulari, per un valore di 585 euro.

- Se la strategia usata dal ladro per riempire lo zaino è di scegliere sempre l'oggetto *di maggior valore* che ci sta, qual è il valore della refurtiva che riuscirà a portare via?

  1 televisore, per un valore di 605 euro.

- Se la strategia usata dal ladro per riempire lo zaino è di scegliere sempre l'oggetto *che ha il miglior rapporto valore/peso* che ci sta, qual è il valore della refurtiva che riuscirà a portare via?

  1 televisore, per un valore di 605 euro.

- Per ottenere il massimo valore della refurtiva, sapete fare meglio?

  2 cellulari, 4 hard disk esterni per un totale di 610 euro.

Problema di ottimizzazione dello zaino

- *Knapsack*: è un problema “di sottoinsieme”.

- Dati (tutti interi positivi; quelli di più oggetti uguali occorrono altrettante volte nelle rispettive liste):

P, peso massimo sopportato dallo zaino,

$(p_1, \ldots, p_n)$ e $(v_1, \ldots, v_n)$, liste di pesi e valori di n oggetti.

- Obiettivo:

determinare un insieme di oggetti il cui peso complessivo sia, al più, P e il cui valore complessivo sia il massimo possibile.

• Un esempio:

$P = 12$

pesi = (1, 2, 3, 3, 5, 6)
valori = (2, 4, 6, 6, 7, 9)

Quindi $n = 6$ oggetti, non importa in quale ordine…

• Quale risultato dà una procedura *greedy*?

Se ogni volta scegliamo l'oggetto di maggior valore possibile, avremo nello zaino gli oggetti 6, 5 e 1:
il peso complessivo sarà 6 + 5 + 1 = 12 (zaino pieno)
e il valore complessivo 9 + 7 + 2 = 18.

Si può fare meglio?

❖ Piccola parentesi 1: algoritmo *greedy*

Ad ogni passo, sceglie l'*ottimo locale* secondo un certo criterio. Esempio: comporre una somma col minimo numero di monete. Con quali tagli funziona?

- Con tagli da 25, 10, 5, 1:

48 = 25 + 10 + 10 + 1 + 1 + 1 — ottimo

45 = 25 + 10 + 10 — ottimo

- Con tagli da 25, 11, 5, 1:

48 = 25 + 11 + 11 + 1 — ottimo

45 = 25 + 11 + 5 + 1 + 1 + 1 + 1 — non ottimo

- Con tagli da 25, 12, 5, 1:

48 = 25 + 12 + 5 + 5 + 1 — non ottimo

45 = 25 + 12 + 5 + 1 + 1 + 1 — non ottimo

Piccola parentesi 1 (continua): con quali tagli funziona?

Caratterizzare i sistemi monetari per i quali l'algoritmo *greedy* funziona è questione irrisolta, se i tagli sono più di cinque…

Si potrebbero generare, una dopo l'altra, tutte le combinazioni che compongono la somma data, scartandole non appena il numero di monete occorrenti supera il minimo finora trovato…

Nella cartella programmi_Python_2.7.9 trovate il programma monete.py, che calcola in quanti modi si forma un importo, disponendo a piacere di monete di vari tagli: provate ad adattarlo al nostro problema!

Due sono le funzioni ivi definite; la seconda, ben più efficiente, sfrutta l'idea della programmazione dinamica …

❖ Piccola parentesi 2: programmazione dinamica

(R. E. Bellman e G. B. Dantzig, 1956-57)

	C1	C2	C3	C4	C5	C6	C7	C8
R8	1	3	7	5	9	11	15	13
R7	0	2	4	6	8	10	12	14
R6	3	5	3	1	7	9	13	11
R5	5	3	4	7	9	0	11	17
R4	3	2	4	6	5	8	10	14
R3	19	15	7	6	8	2	4	5
R2	1	2	5	7	6	3	12	14
R1	15	16	22	15	3	7	9	8

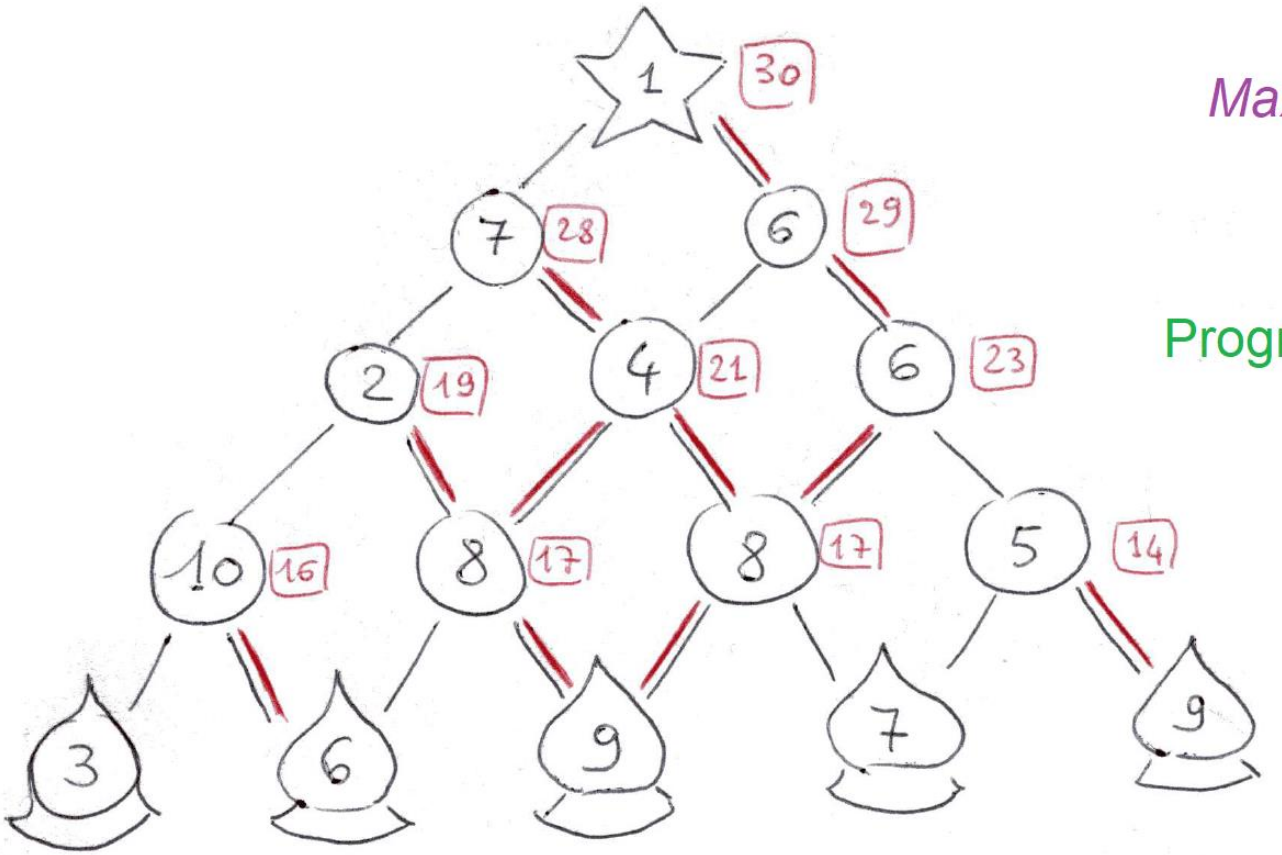
R8	1	3	7	5	9	11	15	13
R7	1a/3d	3s/9d	7s/11a	11a/15d	13s/19d	19s/25d	23s/27a	27a/29s

R8	1	3	7	5	9	11	15	13
R7	1a/3d	3s/9d	7s/11a	11a/15d	13s/19d	19s/25d	23s/27a	27a/29s
R6	4a/12d	6s/16d	6s/18d	8s/20d	18s/32d	22s/36d	32s/42d	34s/40a

. . .

Piccola parentesi 2 (continua): giù dall'albero di Natale!

Dalla stella in cima all'albero, scendere fino a raggiungere una delle campanelle in fondo, in modo da rendere massima la somma dei numeri attribuiti alle decorazioni toccate lungo la discesa.



Problema di
Maximum Sum Descent

Programmazione Dinamica

Codifica (vedi programmi_Python_2.7.9 > discesa.py):

```
p = [[1], [7, 6], [2, 4, 6], [10, 8, 8, 5], [3, 6, 9, 7, 9]]
a = len(p)

# soluzione inefficiente:

def somma(i, j, s):
    if i == a-1: return s
    s1 = somma(i+1, j, s + p[i+1][j])
    s2 = somma(i+1, j+1, s + p[i+1][j+1])
    return max(s1, s2)

print somma(0, 0, p[0][0])

# soluzione efficiente (attenzione: modifica p)

for i in range(a - 2, -1, -1):
    for j in range(0, i + 1):
        p[i][j] += max(p[i+1][j], p[i+1][j+1])
print p[0][0]
```

Zaino ottimo con la programmazione dinamica

• Idea: se in qualche modo abbiamo già parzialmente riempito lo zaino, il maggior profitto lo otterremo comunque massimizzando il valore degli oggetti che vi possono ancora stare, da scegliere tra i rimanenti.

• Ribaltando la prospettiva: supponiamo che la capacità dello zaino aumenti progressivamente da 1 a P, e ad ogni stadio chiediamoci quale sia il massimo valore raggiungibile disponendo soltanto del primo oggetto, o dei primi due, o dei primi tre … o di tutti gli n oggetti.

- Che cosa dobbiamo fare?

Calcolare tutti i numeri M(c, k) per $c = 1, \ldots, P$ e per $k = 1, \ldots, n$, dove M(c, k) è il massimo valore complessivo ottenibile scegliendo dai primi k oggetti, avendo c come limite superiore al peso complessivo.

→ Quanto a strutture di dati, ci servirà una matrice M con almeno P righe e n colonne, oltre a due array p e v di n elementi ciascuno, per i pesi e i valori.

Quando dovremo decidere se scegliere o meno il k-esimo oggetto, ci chiederemo innanzi tutto se il suo peso supera c: se sì, ovviamente non lo potremo scegliere, indipendentemente dal suo valore. Altrimenti, lo sceglieremo soltanto nel caso in cui la sua presenza riesca a migliorare il profitto: dovremo quindi considerare il miglior valore (già calcolato) col peso limite $c - p_k$ e coi primi $k - 1$ oggetti, e aggiungervi il valore v_k, per poter prendere di conseguenza la giusta decisione.

→ È quindi opportuno prevedere anche una riga 0 e una colonna 0 nella matrice M …

- L'algoritmo

L'algoritmo usa una matrice di interi M, di $P+1$ righe per $n+1$ colonne, dove la prima riga e la prima colonna (che supponiamo abbiano indice 0) sono inizializzate col valore 0: infatti, se lo zaino non può contenere nulla o non vi è alcun oggetto, allora il suo valore (ottimo) è 0.

per $c = 1, \ldots, P$:
  per $k = 1, \ldots, n$:
    se $p_k > c$ **allora**
      $\mathrm{M}(c, k) \leftarrow \mathrm{M}(c, k-1)$
    altrimenti
      $\mathrm{M}(c, k) \leftarrow max\{\, \mathrm{M}(c - p_k, k-1) + v_k\,,\ \mathrm{M}(c, k-1)\,\}$

Fatto questo, nell'elemento in ultima riga e ultima colonna, cioè M(P, n), è contenuto il valore di una soluzione ottima.

- Riprendiamo l'esempio:

$P = 12, \quad p = (1, 2, 3, 3, 5, 6), \quad v = (2, 4, 6, 6, 7, 9)$

Otteniamo la seguente matrice M, con M(12, 6) = 21:

0	0	0	0	0	0	0
0	2	2	2	2	2	2
0	2	4	4	4	4	4
0	2	6	6	6	6	6
0	2	6	8	8	8	8
0	2	6	10	10	10	10
0	2	6	12	12	12	12
0	2	6	12	14	14	14
0	2	6	12	16	16	16
0	2	6	12	18	18	18
0	2	6	12	18	18	18
0	2	6	12	18	19	19
0	2	6	12	18	21	21

- Come risalire agli oggetti da mettere nello zaino?

$P = 12, \quad p = (1, 2, 3, 3, 5, 6), \quad v = (2, 4, 6, 6, 7, 9)$

0	0	0	0	0	0	0
0	2	2	2	2	2	2
0	2	4	4	4	4	4
0	2	6	6	6	6	6
0	2	6	8	8	8	8
0	2	6	10	10	10	10
0	2	6	12	12	12	12
0	2	6	12	14	14	14
0	2	6	12	16	16	16
0	2	6	12	18	18	18
0	2	6	12	18	18	18
0	2	6	12	18	19	19
0	2	6	12	18	21	21

- Partiamo dall'elemento in basso a destra (che contiene il valore ottimo) e, restando sull'ultima riga, spostiamoci a sinistra, fino a incontrare una variazione di valore:
→ tra le colonne 5 e 4: prendiamo l'oggetto 5 e togliamo il suo peso da 12 → 12 – 5 = 7 è il peso massimo rimanente.
- Risaliamo la colonna 4 fino alla riga 7 e poi spostiamoci a sinistra, fino a incontrare una variazione di valore:
→ c'è subito, tra le colonne 4 e 3: prendiamo l'oggetto 4 e togliamo il suo peso da 7 → 7 – 3 = 4 è il peso residuo.
- Risaliamo la colonna 3 fino alla riga 4 e poi spostiamoci a sinistra, fino a incontrare una variazione di valore:
→ c'è subito, tra le colonne 3 e 2: prendiamo l'oggetto 3 e togliamo il suo peso da 4 → 4 – 3 = 1 è il peso residuo.
- Risaliamo la colonna 2 fino alla riga 1 e poi spostiamoci a sinistra: il valore varia tra le colonne 1 e 0, per cui prendiamo l'oggetto 1 e togliamo il suo peso da 1 → 1 – 1 = 0.

Scegliendo gli oggetti 5, 4, 3 e 1, il valore è massimo e, in questo caso, lo zaino è stato riempito…

Tuttavia, vi sono altre soluzioni ugualmente ottime: quali? Quella che si trova dipende dall'ordine in cui si dispongono inizialmente gli oggetti…

- Scriviamo la seconda parte dell'algoritmo:

$c \leftarrow P$; $k \leftarrow n$;
finché $c > 0$ **e** $k > 0$:
    se $\mathrm{M}(c, k) \neq \mathrm{M}(c, k-1)$ **allora**
        è scelto l'oggetto k
        $c \leftarrow c - p_k$
    $k \leftarrow k - 1$

- Un caso particolare: valori uguali ai pesi

Se lo scopo è quello di riempire lo zaino il più possibile, pur di non superare la capacità P, basta far coincidere i valori degli oggetti con i rispettivi pesi: si pensi, ad esempio, a dei lingotti d'oro di diverse dimensioni. Si provi a risolvere questo problema, con uno zaino di capacità 150 (unità di peso) e otto lingotti di peso 16, 27, 37, 42, 52, 59, 65 e 95: quali dovranno essere scelti? (Qui si riesce a riempire lo zaino al massimo del peso sopportato; ma se la sua capacità scendesse a 149, allora la soluzione ottima lo riempirebbe *quasi* completamente, scegliendo lingotti tutti diversi dall'istanza precedente…)

Nota: nella formulazione originaria del problema decisionale (R. M. Karp, 1972), parlando soltanto di pesi, si chiedeva di stabilire se lo zaino può essere riempito esattamente col suo peso massimo P, ciò che equivale a chiedere se l'equazione

$$p_1 x_1 + p_2 x_2 + \ldots + p_n x_n = P$$

ha soluzioni in cui ciascuna incognita vale 0 o 1.

→ problema di *Subset-Sum* (… si trova in crittografia)

Porsi la stessa domanda per un sistema lineare a coefficienti interi non cambia la complessità del problema, che viene detto di

programmazione (lineare) intera 0-1

□

- Una variante: il problema con ripetizioni

Una variante significativa del problema sottintende l'illimitata disponibilità di esemplari di ciascun tipo di oggetto.

Questo problema, noto come problema dello zaino *con ripetizioni*, per essere risolto richiede uno spazio di memoria inferiore, ma l'ordine di grandezza temporale non cambia. È sufficiente una sola colonna della matrice usata nel caso precedente, e quindi diciamo che adesso M è soltanto un *array* di $P+1$ interi…

Ci servirà però un altro array dello stesso formato…

- Complessità rispetto al tempo

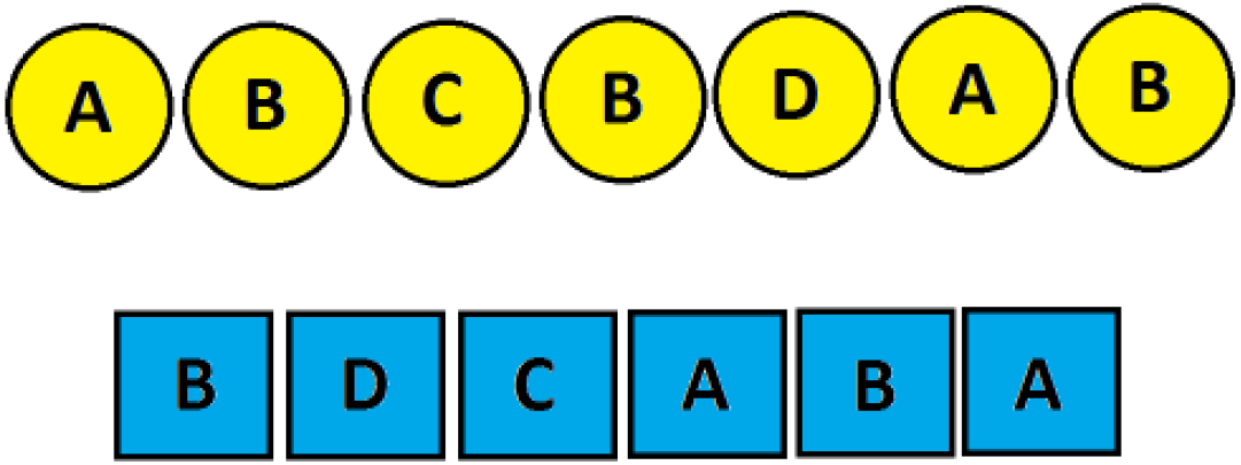
Anche per questa variante, si giunge a un algoritmo che richiede un tempo di esecuzione dell'ordine di $n \cdot P$ (per i due cicli annidati), che ***non*** è polinomiale nella dimensione dell'input, costituito da circa

$$(n+1)\cdot\log_2 P + n\cdot\log_2 v_{\max}$$

bit. Quando l'input è una lista di numeri e il tempo di esecuzione è limitato da un polinomio nel maggiore di tali numeri e nella lunghezza della lista, si parla di algoritmi pseudo-polinomiali.

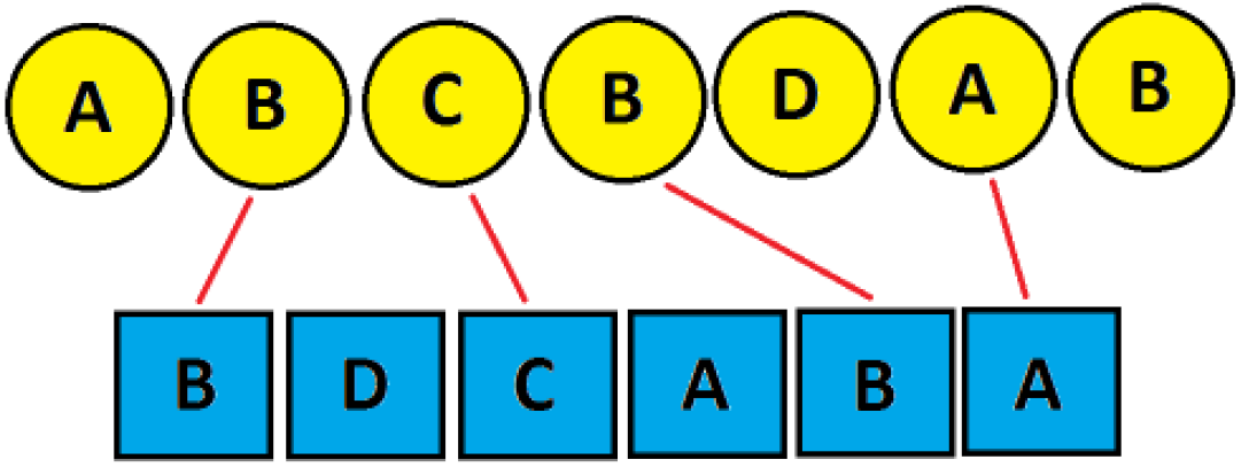
Il problema dello zaino è *NP-hard* ma non *strongly*!

More Candy (Canada, 2013)



You can connect a circle and a square with the same letter in it by clicking on them.
Try to make as many connections as possible, without crossing the lines!

Answer



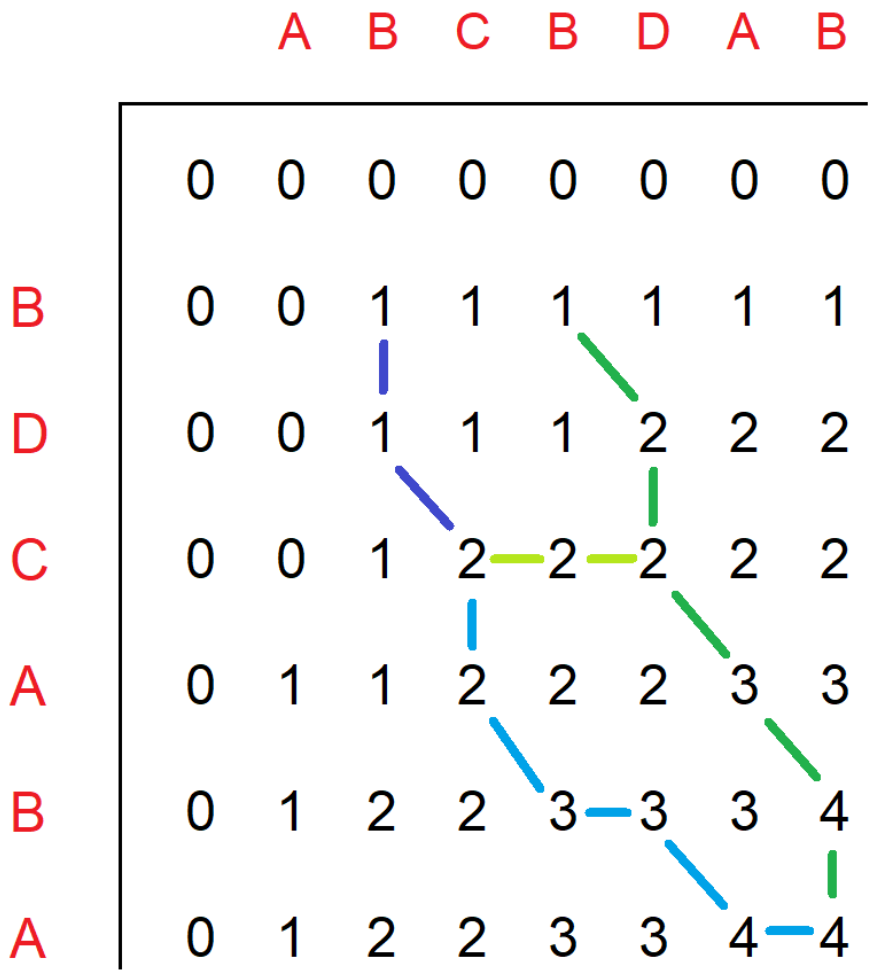
Il problema della *longest common-subsequence*

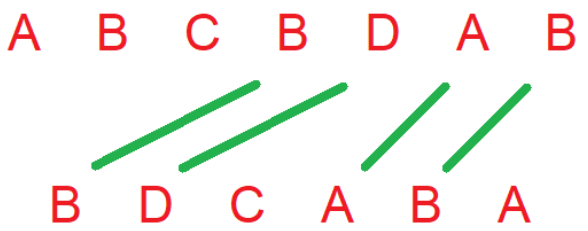
Vedi programmi_Python_2.7.9 > lcs_fb.py (questo) e lcs_pd.py (il successivo)

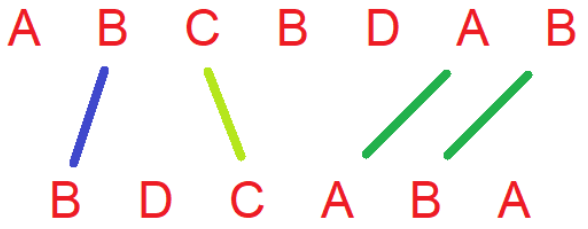
```
# Risolve il problema LCS con la forza bruta
# Complessita' rispetto al tempo in O(2^(m * n))

def lcs(X, Y, m, n):
    if m == 0 or n == 0:
        return 0
    elif X[m-1] == Y[n-1]:
        return lcs(X, Y, m-1, n-1) + 1
    else:
        return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n))

S1 = "BDCABA"
S2 = "ABCBDAB"
print "lunghezza della LCS =", lcs(S1, S2, len(S1), len(S2))
```

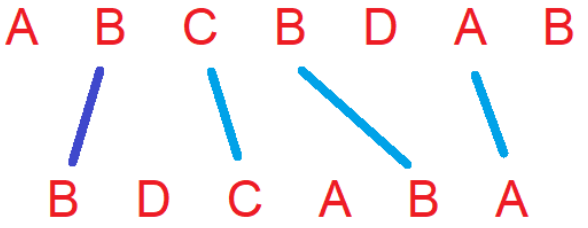
Una soluzione più efficiente per il problema della *lcs*

```
# Risolve il problema LCS con la programmazione dinamica
# Complessita' rispetto al tempo in O(m * n)

def lcs (X, Y, m, n):
    # alloca la matrice L di (m+1) righe per (n+1) colonne
    # e assegna 0 a tutti i suoi elementi
    L = [[0]*(n+1) for i in range(m+1)]

    # L[i][j] contiene la lunghezza della LCS di X[0..i-1] e Y[0..j-1]
    for i in range(1, m+1):
        for j in range(1, n+1):
            if X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1] + 1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])

    # L[m][n] contiene la lunghezza della LCS di X[0..n-1] e Y[0..m-1]
    for i in range(m+1):
        for j in range(n+1): print L[i][j],
        print "\n"
    return L[m][n]

S1 = "BDCABA"
S2 = "ABCBDAB"
ris = lcs(S1, S2, len(S1), len(S2))
print "lunghezza della LCS =", ris
```

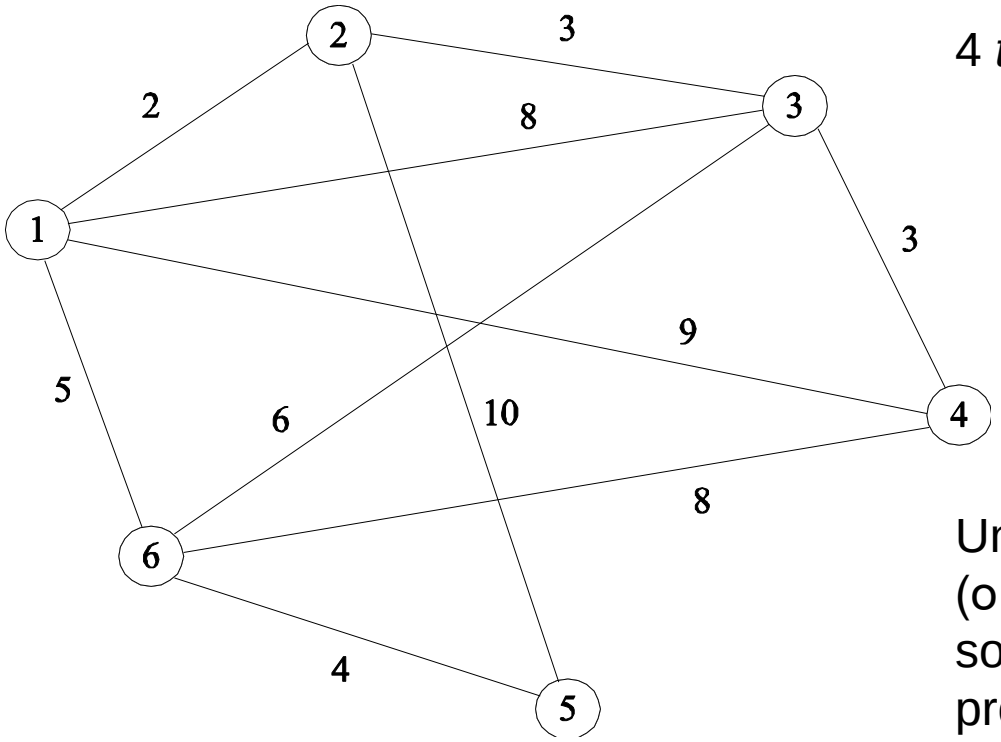

A B C B D A B
0 0 0 0 0 0 0 0
B 0 0 1 1 1 1 1 1
D 0 0 1 1 1 2 2 2
C 0 0 1 2 2 2 2 2
A 0 1 1 2 2 2 3 3
B 0 1 2 2 3 3 3 4
A 0 1 2 2 3 3 4 4


A B C B D A B
B D C A B A


A B C B D A B
B D C A B A


A B C B D A B
B D C A B A

Il problema del commesso viaggiatore (TSP)



4 tour (cicli hamiltoniani) possibili:

[1, 2, 5, 6, 3, 4, 1] con costo 34;
[1, 4, 3, 2, 5, 6, 1] con costo 34;
[1, 2, 5, 6, 4, 3, 1] con costo 35;
[1, 3, 2, 5, 6, 4, 1] con costo 42.

Uno dei primi due indifferentemente (o il suo “rovescio”) costituisce la soluzione di questa istanza del problema.

Non si sa se sia intrinsecamente esponenziale, ma finora non si è trovato alcun algoritmo efficiente per risolverlo in generale!

- In generale, nessun algoritmo di approssimazione, ma algoritmi spesso praticabili

  1954: Dantzig, Fulkerson e Johnson, su 42 città degli USA: programmazione lineare (Dantzig, 1947)

  1962: Held e Karp: progr. dinamica (Bellman, 1957)
  miglior tempo di esecuzione nel caso peggiore: $O(n^2 \cdot 2^n)$, decisamente meglio di $O(n!)$,
  ma anche lo spazio cresce esponenzialmente con n

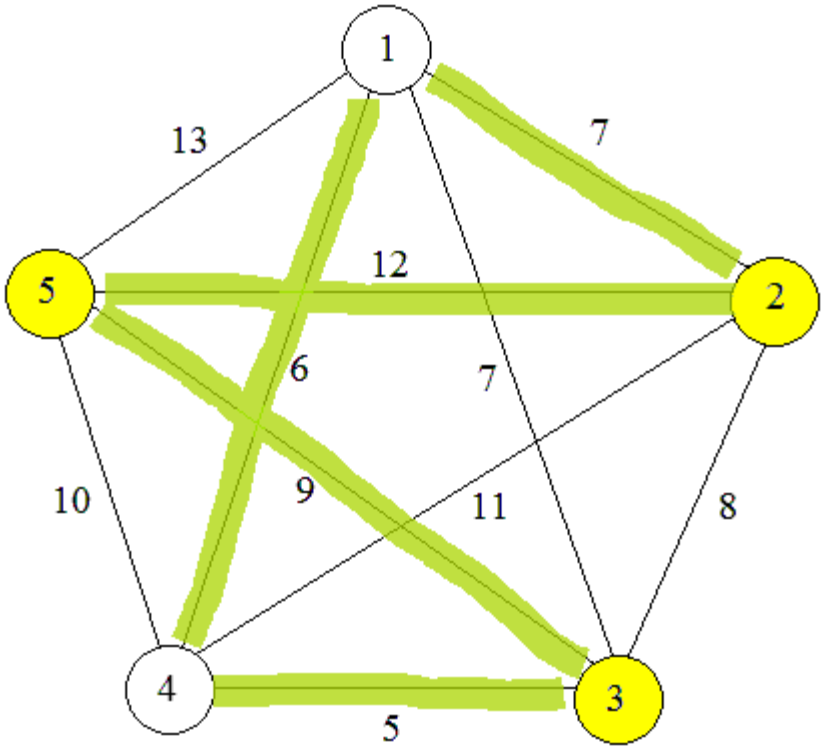
  1963: Little *et al.*: branch-and-bound (Land e Doig, 1960)

- Casi particolari

  TSP metrico: $C(i, j) \leq C(i, k) + C(k, j) \quad \forall\, i, j, k$
  algoritmi di approssimazione (con fattori 2, 3/2)

  TSP euclideo: i nodi sono punti del piano, i costi le distanze
  algoritmo esatto in tempo sub-esponenziale

TSP: un esempio con grafo completo, metrico (e simmetrico)



Tour ottimo:

[1, 2, 5, 3, 4, 1] con costo 39

(In questo piccolo esempio, funziona persino l'algoritmo *greedy*, ammesso che il nodo di partenza sia 2, 3 o 5 …)

I tanti algoritmi studiati per il TSP "funzionano bene" nella maggior parte delle usuali applicazioni …

I "casi peggiori" non sono poi così frequenti nella realtà!

Si vedano i più recenti successi al sito http://www.tsp.gatech.edu/

TSP: l'algoritmo più semplice (anche su grafi orientati)

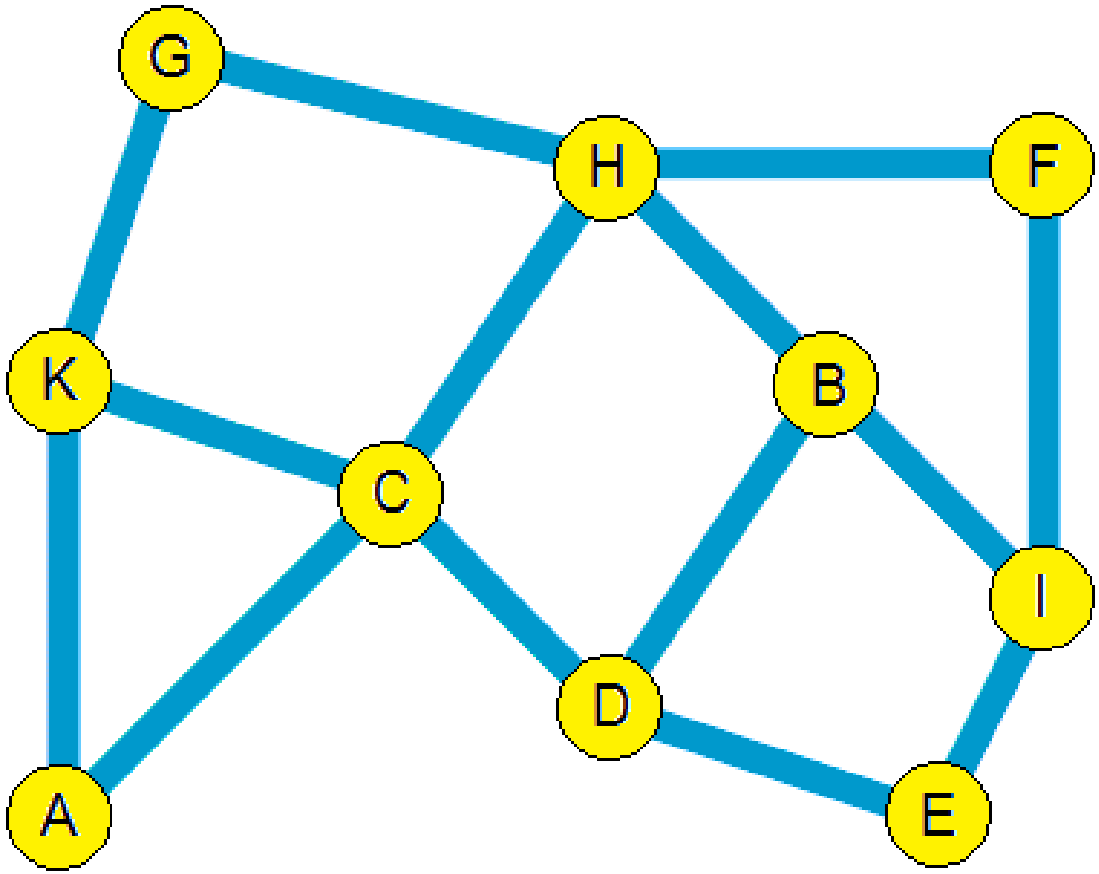
```
TSP (percorso_corrente, costo_corrente):
    i ← ultimo nodo di percorso_corrente
    non_visitati ← lista dei nodi del grafo eccetto quelli elencati in percorso_corrente
    se non_visitati contiene almeno un nodo allora
        per ogni j contenuto in non_visitati e tale che esista l'arco da i a j
            c ← costo_corrente + C(i, j)
    (*)     se c < costo_minimo allora
                tentativo ← percorso_corrente + j (aggiunto come ultimo)
                TSP (tentativo, c)  // chiamata ricorsiva
    altrimenti
        // non_visitati è vuota, ossia percorso_corrente contiene tutti i nodi del grafo
        se esiste l'arco da i a 1 e costo_corrente + C(i, 1) < costo_minimo allora
            cammino_minimo ← percorso_corrente + 1 (aggiunto come ultimo)
            costo_minimo ← costo_corrente + C(i, 1)
```

```
C = matrice dei costi   (+∞ = assenza di arco)
cammino_minimo ← [1]
costo_minimo ← +∞
TSP ([1], 0)
```

---

test (*): può evitare che si esplorino proprio *tutti* i possibili percorsi ...

Dove installare le torrette di avvistamento?



Problema di minima copertura per nodi

- *Minimum Vertex Cover*: problema “di sottoinsieme”.

- Dati:

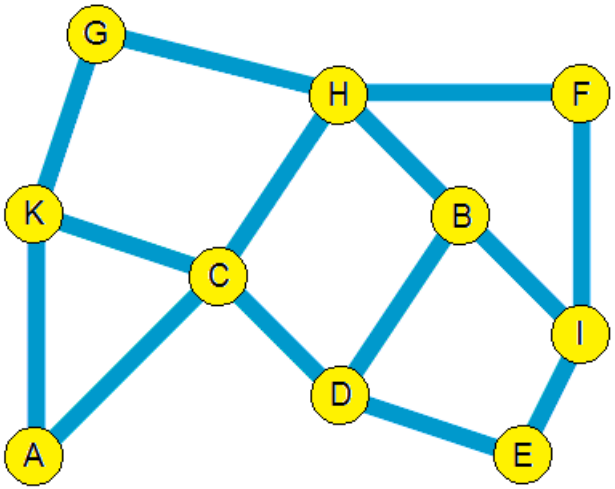
un grafo semplice, non orientato, non pesato, privo di cappi, connesso → può essere rappresentato da una lista di coppie (u, v) con $u < v$, denotando gli n nodi con i numeri da 1 a n.

- Obiettivo:

determinare un insieme di nodi di cardinalità minima tale che ogni arco del grafo abbia in tale insieme almeno uno dei due nodi estremi.

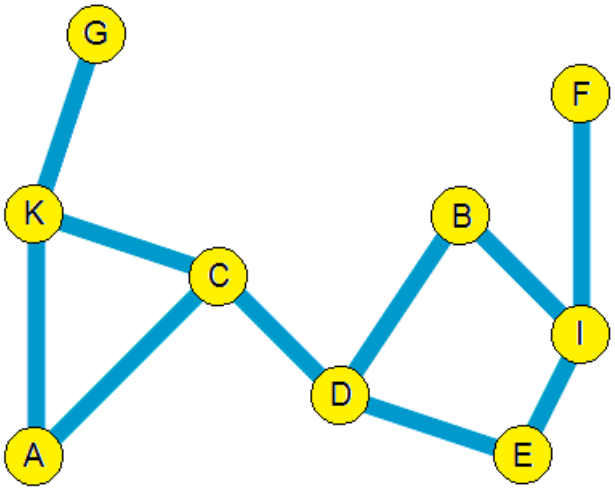
• Un primo algoritmo *greedy*

Idea: per costruire l'insieme copertura X (inizialmente vuoto), ad ogni passo scegliamo un nodo *v* di grado massimo, aggiungiamo *v* a X, e togliamo dal grafo sia *v* sia gli archi che hanno un estremo in *v* (e anche gli eventuali nodi che rimangono isolati) …



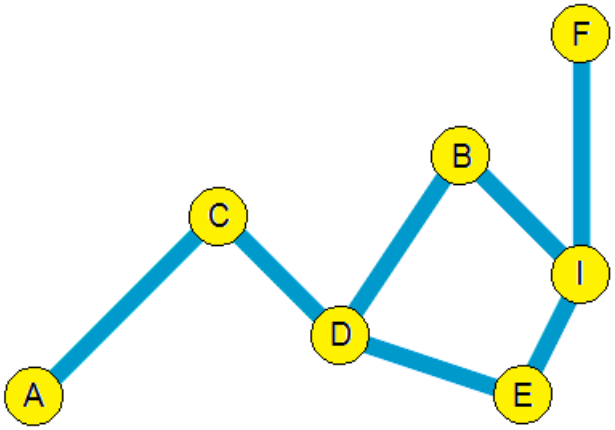
Al primo passo abbiamo due alternative: o C o H (con 4 archi incidenti).

Scegliamo H …



Al secondo passo, i nodi di grado massimo (3) sono quattro: K, C, D, I.

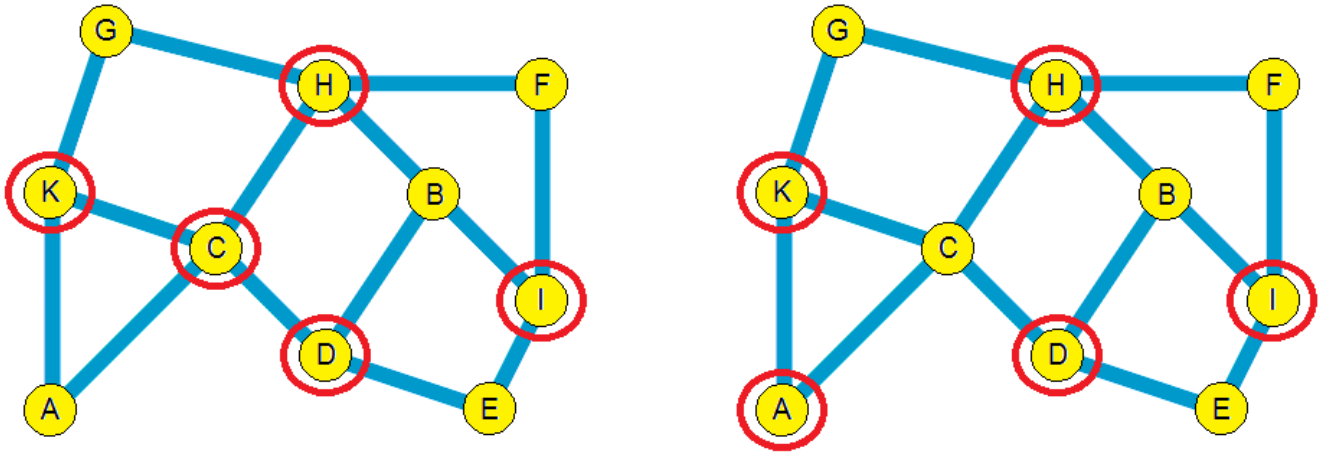
Scegliamo K …



Al terzo passo restano due alternative: D e I.

…

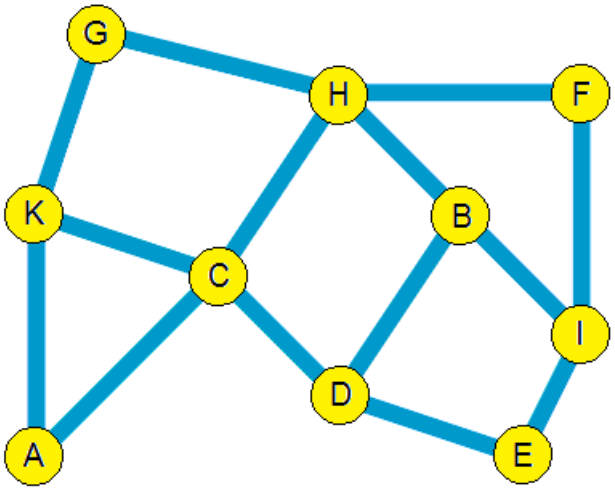
Partendo con H, qualunque successiva scelta, a parità di grado massimo, porta a una delle due coperture minime.



Ma scegliendo inizialmente C e poi B (di grado 3)…
→ Non sempre funziona, anzi non è nemmeno un algoritmo di approssimazione con fattore costante: nei casi pessimi, il rapporto (num. nodi scelti / num. nodi ottimi) tende a crescere (in modo logaritmico) all'aumentare del num. di nodi del grafo.

• Un secondo algoritmo *greedy*

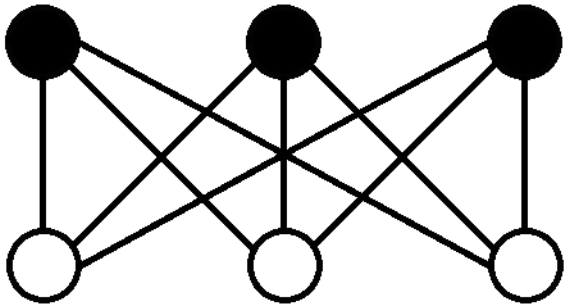
Idea: per ogni arco, almeno uno dei due nodi estremi deve stare in una copertura minima. Quindi, per costruire l'insieme X, ad ogni passo scegliamo (a caso) un arco, diciamo quello tra i due nodi *u* e *v*, aggiungiamo a X sia *u* sia *v*, e togliamo dal grafo *u*, *v* e ogni arco incidente su *u* o su *v* …



In questo esempio, nessuna scelta porta a una soluzione ottima… e se siamo sfortunati prendiamo tutti i nodi!

In generale, nel caso peggiore, la copertura generata è grande il doppio della minima: è dunque un algoritmo di approssimazione con fattore costante 2.

Ad esempio, nel caso di $K_{n,n}$ sono presi tutti i $2n$ nodi, mentre le due coperture minime ne contengono n.
Con $n = 3$:

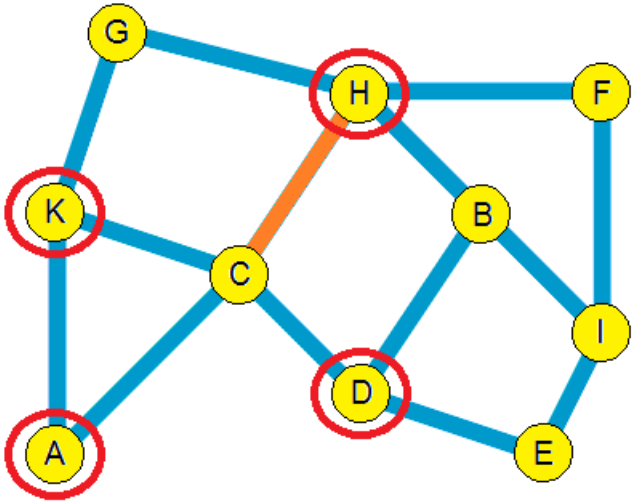


Nota: si tenga presente che su qualsiasi grafo bipartito il problema è risolubile (esattamente) in modo efficiente.

□

• Abbinare i due criteri, scegliendo un arco non a caso, ma con la massima somma dei gradi dei due nodi estremi?
Nel caso di $K_{n,n}$ non vi sarebbe alcun beneficio!

• Idea per trovare una soluzione ottima: modifichiamo il secondo algoritmo *greedy*, calcolando **anche** che cosa si ottiene aggiungendo a X o soltanto *u* o soltanto *v*, tenendo presente che, quando si aggiunge uno solo dei due, si devono per forza aggiungere nel contempo anche tutti i nodi adiacenti all'altro.



Ad esempio, scegliendo H ed escludendo C, è necessario includere D, A, K.

Un algoritmo per ottenere una soluzione ottima

- Se G è privo di archi, allora restituisce l'insieme vuoto. Fine.
- Altrimenti, sia (u, v) un arco di G, arbitrariamente scelto;
  ricorsivamente, risolve tre problemi (possibilmente in parallelo):

1. $X \leftarrow \{w \mid (w, u)$ è un arco di $G\}$ e sia G_1 il grafo che si ottiene da G togliendovi i nodi in X e gli archi su di essi incidenti; sia X_1 la soluzione trovata quando è dato G_1; infine, $X_1 \leftarrow X_1 \cup X$.
2. $X \leftarrow \{w \mid (w, v)$ è un arco di $G\}$ e sia G_2 il grafo che si ottiene da G togliendovi i nodi in X e gli archi su di essi incidenti; sia X_2 la soluzione trovata quando è dato G_2; infine, $X_2 \leftarrow X_2 \cup X$.
3. Sia G_3 il grafo che si ottiene da G togliendovi i nodi u e v e gli archi su di essi incidenti; sia X_3 la soluzione trovata quando è dato G_3; infine, $X_3 \leftarrow X_3 \cup \{u, v\}$.

- Restituisce l'insieme di cardinalità minore tra X_1, X_2 e X_3. Fine.

- Complessità rispetto al tempo

Si dimostra che, se k è il numero di nodi della minima copertura, allora la profondità a cui giunge l'algoritmo ora descritto è, al più, $k \rightarrow$ la complessità rispetto al tempo presenta un fattore 3^k.

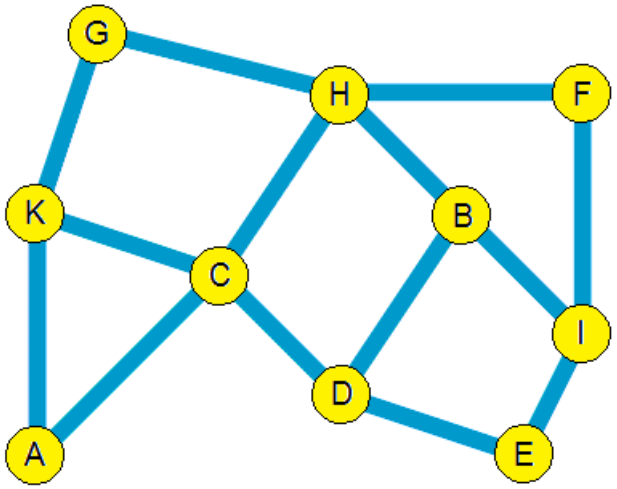
In effetti, il problema è *NP-hard*.

- È un caso particolare di minima copertura di un insieme (*Minimum Set Cover*): basta considerare, per ogni nodo, l'insieme degli archi su di esso incidenti e, come "universo", l'insieme di tutti gli archi del grafo.

Minimum Set Cover è *equivalente* al minimo insieme dominante (*Minimum Dominating Set*).

• Come ricondurre efficientemente MDS a MSC?

Per ogni nodo *v*, consideriamo l'insieme costituito da *v* e tutti i nodi adiacenti a *v* …



{ A, C, K }
{ C, A, D, H, K }
{ E, D, I }
{ G, H, K }
{ I, B, E, F }

{ B, D, H, I }
{ D, B, C, E }
{ F, H, I }
{ H, B, C, F, G }
{ K, A, C, G }

Qui ci sono ben 8 coperture di soli tre sottoinsiemi (2 sono evidenziate in rosso e in verde, rispettivamente).
Nota: nessuna copertura è "esatta".

Provate a ricondurre Min. Set Cover a Min. Dominating Set.

“Complementare”: massimo insieme indipendente

Trovare un insieme di nodi di cardinalità massima che, a due a due, *non* sono collegati da un arco.

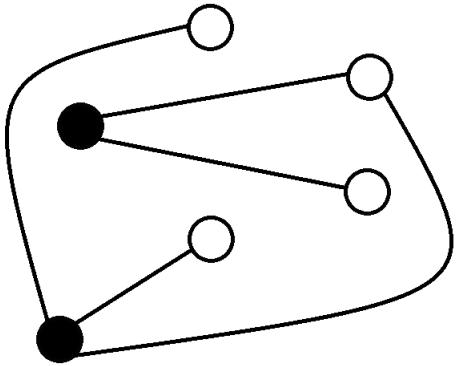
Se X è un insieme di nodi, sono equivalenti:

- X è un insieme indipendente
- ogni arco incide su al più un nodo $\in$ X
- ogni arco incide su almeno un nodo $\notin$ X
- il complemento di X è una copertura per nodi

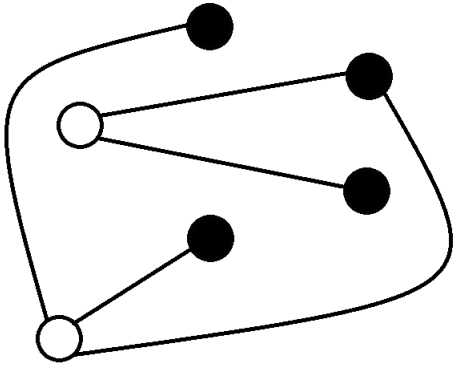
$\rightarrow$ un massimo insieme indipendente è complemento di una minima copertura per nodi.

Considerando poi il grafo complementare …

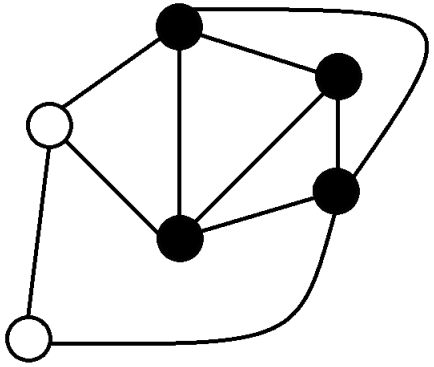
Un insieme indipendente è costituito dai nodi di un sottografo completo (*clique*) nel grafo complementare.



Minimum Vertex Cover



Maximum Independent Vertex Set



Maximum Clique (nel grafo complementare)

Algoritmi esatti

Se X è un massimo insieme indipendente, allora, per ogni nodo *v* del grafo, delle due l'una: o *v* o almeno uno dei suoi adiacenti appartiene a X …

→ per tutti i citati problemi "difficili" sui grafi, si può pensare di risolvere, ricorsivamente e possibilmente in parallelo, due o più sottoproblemi di dimensione ridotta, e poi confezionare una soluzione esatta utilizzando una di quelle dei sottoproblemi che presenti determinate caratteristiche

→ tuttavia, il tempo cresce esponenzialmente col numero di nodi …

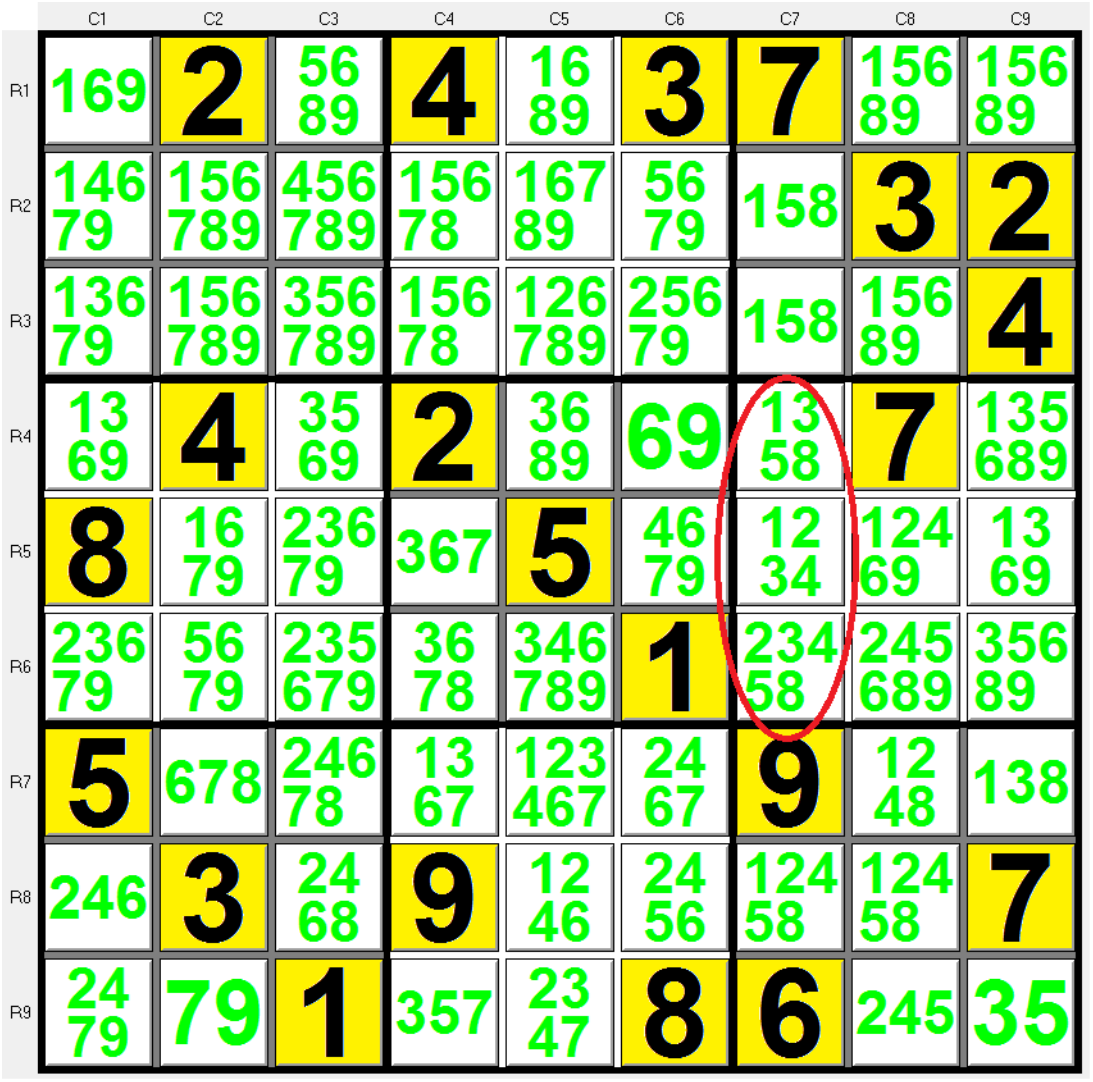
❑ Puzzle semplici da programmare (con *backtracking*)

- Una soluzione, se c'è, è tutta contenuta nello "stato finale", non nella sequenza di mosse per raggiungerlo.
- Per evitare di incorrere in cicli, se c'è questo pericolo, basta lasciare nello stato corrente una traccia del "percorso" fatto (che alla fine indicherà la soluzione trovata).

Esempi (trovare una soluzione, se c'è, oppure tutte …):

- completamento di uno schema di Sudoku (anche come istanza di *Exact Cover Set* o, in parte, con tecniche *ad hoc*)
- puzzle con tessere a incastro (ad es. polimini …)
- disegno di un giro di cavallo (aperto o chiuso) su scacchiera $n \times n$ o $m \times n$, oppure di un percorso per uscire da un labirinto

Uno schema di Sudoku assai difficile



	C1	C2	C3	C4	C5	C6	C7	C8	C9
R1	169	2	56 89	4	16 89	3	7	156 89	156 89
R2	146 79	156 789	456 789	156 78	167 89	56 79	158	3	2
R3	136 79	156 789	356 789	156 78	126 789	256 79	158	156 89	4
R4	13 69	4	35 69	2	36 89	69	13 58	7	135 689
R5	8	16 79	236 79	367	5	46 79	12 34	124 69	13 69
R6	236 79	56 79	235 679	36 78	346 789	1	234 58	245 689	356 89
R7	5	678	246 78	13 67	123 467	24 67	9	12 48	138
R8	246	3	24 68	9	12 46	24 56	124 58	124 58	7
R9	24 79	79	1	357	23 47	8	6	245	35

- Intersezione tra sesto riquadro e C7: il numero 3 non appare altrove in C7

e quindi 3 si può cancellare dalle altre caselle del sesto riquadro (R4C9, R5C9 e R6C9) … Ma poi?

"The Sudoku Susser" qui si arrende, e lo risolve soltanto con la "forza bruta"!

*** Per trovare una soluzione di uno schema S o concludere che non ne esistono:**

```
funzione risolvi lo schema S
    se in S esiste una casella dove non può stare alcun numero
    allora termina con "insuccesso";
    altrimenti
        se in S non esiste alcuna casella dove possano stare almeno due numeri
        allora
            stampa lo schema S;
            termina con "successo";
        altrimenti
            sia RiCj una casella di S in cui possono stare almeno due numeri; (*)
            per ogni numero N che può stare in RiCj
                fai una copia S' dello schema S;
                lascia soltanto N nella casella RiCj di S' e cancella N dalle
                altre caselle di stesso riquadro/riga/colonna (e, ricorsivamente,
                ogni volta che rimane un solo numero in una casella, cancellalo
                dalle altre caselle di stesso riquadro/riga/colonna);
                se risolvi lo schema S' termina con "successo"
                allora termina con "successo";
            termina con "insuccesso";
```

Si osservi che si tratta di una funzione con un effetto collaterale: la stampa della soluzione trovata!

(*) Per tentare di rendere minimo il numero di tentativi, converrebbe scegliere una delle caselle col minor numero (> 1) di possibilità - ma non è garantito!

```
* Per trovare tutte le soluzioni di uno schema S:

procedura stampa le soluzioni dello schema S
    se in ogni casella di S può stare almeno un numero
    allora
        se in S non esiste alcuna casella dove possano stare almeno due numeri
        allora
            stampa lo schema S;
        altrimenti
            sia RiCj una casella di S in cui possono stare almeno due numeri; (*)
            per ogni numero N che può stare in RiCj
                fai una copia S' dello schema S;
                lascia soltanto N nella casella RiCj di S' e cancella N dalle
                altre caselle di stesso riquadro/riga/colonna (e, ricorsivamente,
                ogni volta che rimane un solo numero in una casella, cancellalo
                dalle altre caselle di stesso riquadro/riga/colonna);
                stampa le soluzioni dello schema S';
```

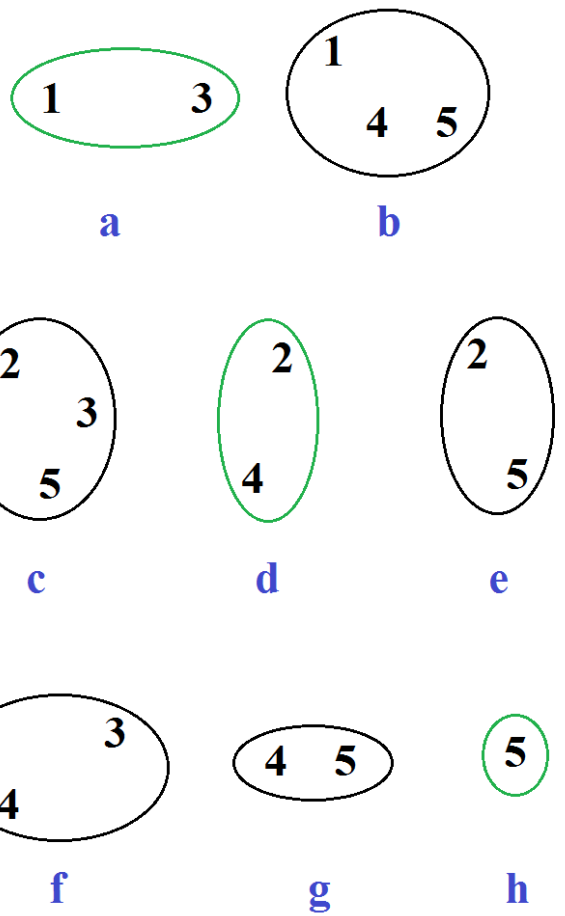
Questa definizione è ancora più semplice di quella della precedente funzione!
(*) Per tentare di rendere minimo il numero di tentativi, converrebbe scegliere una delle caselle col minor numero (> 1) di possibilità - ma non è garantito!

- Come dice Jean-Paul Delahaye, questo procedimento, a mano, "non è praticabile, perché richiederebbe una pazienza sovrumana". Almeno nell'ambito del Sudoku tradizionale, "il metodo più efficiente per una macchina è il più faticoso per un essere umano".

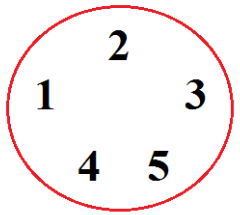
Exact cover problem

- Problema di “esatta copertura” di un insieme (strongly NP-hard): un esempio.

“Se possibile, scegliere degli insiemi tra questi otto, in modo che sia preso una e una sola volta ciascun elemento di S.”



S =

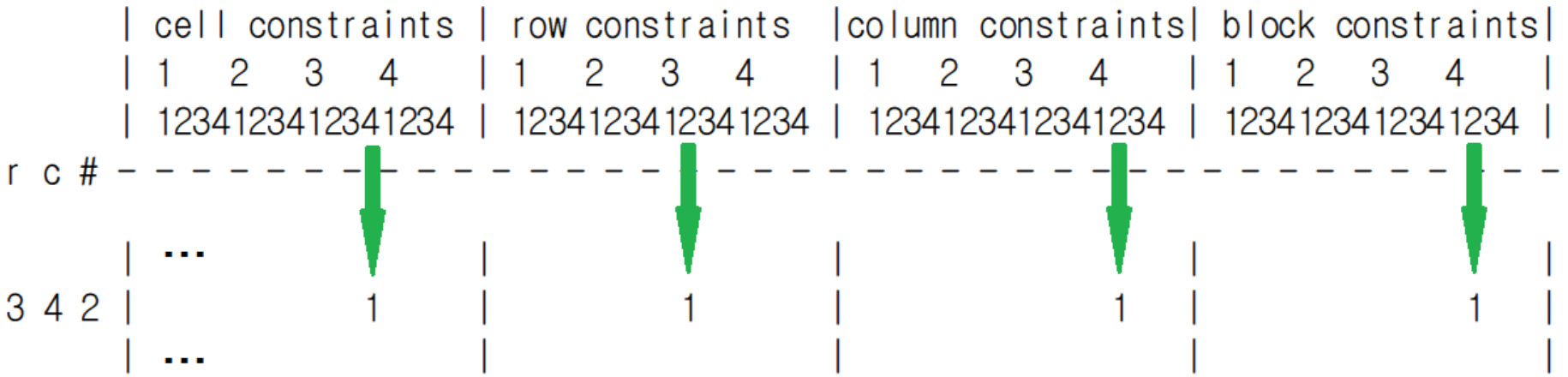


	1	2	3	4	5
a	1	0	1	0	0
b	1	0	0	1	1
c	0	1	1	0	1
d	0	1	0	1	0
e	0	1	0	0	1
f	0	0	1	1	0
g	0	0	0	1	1
h	0	0	0	0	1

Sudoku come *exact cover problem* …

- Un puzzle di Sudoku può ricondursi a un'istanza di tale problema.

Ad esempio, nel caso 4 x 4:



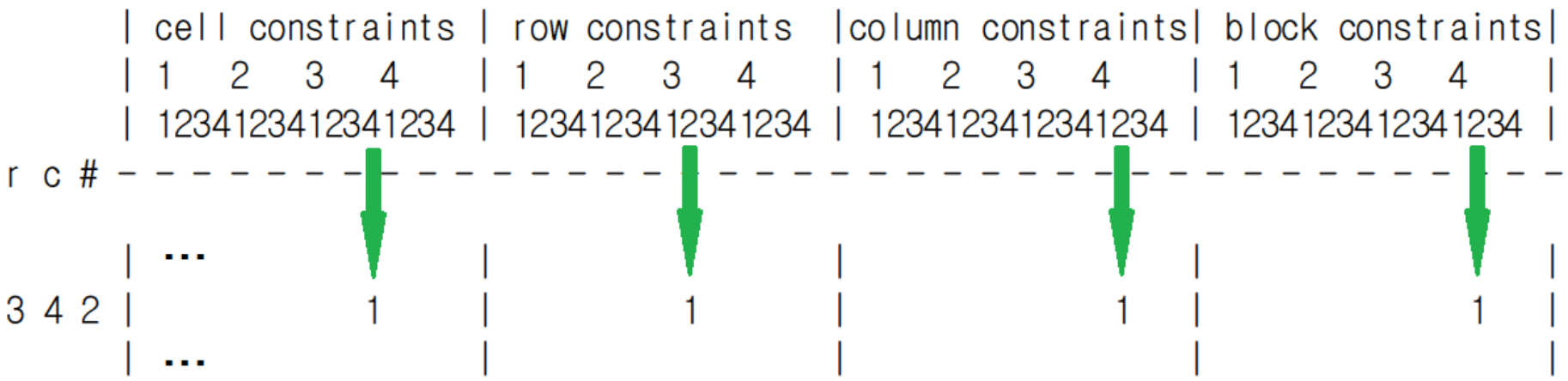
34	1	23	234
2	4	13	134
1	2	4	123
14	3	12	12

… in R3C4 può stare il 2; se lo fissiamo:

- tale casella non è più occupabile,
- nella terza riga c'è il 2,
- nella quarta colonna c'è il 2,
- nel quarto riquadro c'è il 2.

Quante righe ha questa matrice?

- Nel caso 4 x 4 (k^2 x k^2, con k = 2):



Matrice di bit, con 64 righe (quindi quadrata, ma è un caso!) …
Ogni numero fissato nello schema iniziale esclude tuttavia 3 righe!

- In generale: matrice di bit, con $4 \cdot k^4$ colonne e $k^6 – (k^2 – 1) \cdot n$ righe, se n sono i numeri scritti nello schema iniziale;
ciascuna riga contiene comunque quattro 1.
Nel Sudoku classico, k = 3; ma risolvere Sudoku (k) è NP-hard …

- *Problema:* determinare un sottoinsieme di k^4 righe che presentino esattamente un 1 in ciascuna colonna.

… e metodi per risolverlo

- Knuth: *Algorithm X* per *exact cover problem*, basato su back-tracking e realizzato con una tecnica particolare (*dancing links*), è piuttosto efficiente per istanze di ragionevole dimensione.

Donald E. Knuth, Stanford University, 2000: “Dancing Links”
http://www-cs-faculty.stanford.edu/~knuth/papers/dancing-color.ps.gz

- Realizzato in Python (ma usando sets anziché liste bidirezionali circolari) e applicato al Sudoku 9 x 9, richiede qualche centesimo di secondo, al più alcuni secondi per i puzzle più difficili.

Ali Assaf, 2013: “Algorithm X in 30 lines!”
http://www.cs.mcgill.ca/~aassaf9/python/algorithm_x.html

- Per non perdere lo spirito del gioco, si possono combinare backtracking e strategie logiche elementari, facilmente calcolabili: l'efficiente programma di Norvig, scritto in Python, è basato su due funzioni mutuamente ricorsive.

  Peter Norvig, 2011: "Solving every Sudoku puzzle"
  http://www.norvig.com/sudoku.html

- In altri interessanti articoli, Algorithm X è spiegato, realizzato e applicato a diversi rompicapi: pentamini e loro varianti (*Kanoodle*), quadrati latini, Sudoku e creazione di nuovi schemi iniziali …

  Andrzej Kapanowski, "Python for Education: The Exact Cover Problem", The Python Papers 6, 2 (2011)
  http://ojs.pythonpapers.org/index.php/tpp/article/view/227

  David Austin, AMS, 2015: "Puzzling Over Exact Cover Problems"
  http://www.ams.org/samplings/feature-column/fcarc-kanoodle

  Team # 3140, "The Application of Exact Cover to the Creating of Sudoku Puzzle"
  http://www.math.utah.edu/~yzhang/teaching/1030/Sudoku.pdf

Un programma semplice per costruire schemi

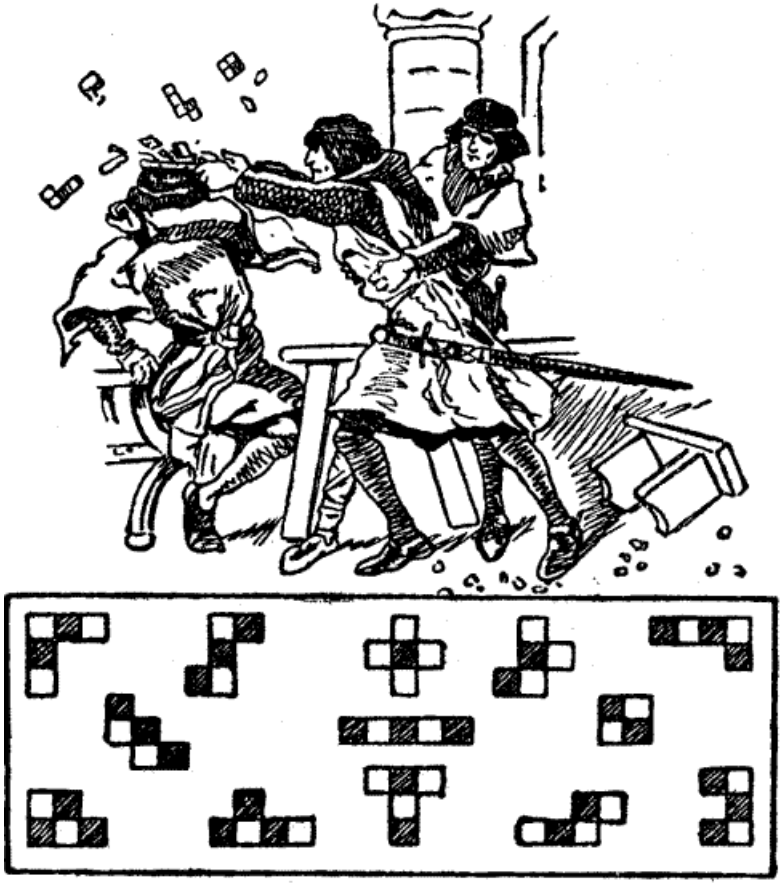
*** Per creare uno schema iniziale che ammetta una e una sola soluzione:**

```
disponi "a caso" alcuni numeri in uno schema S, inizialmente vuoto;
stampa le soluzioni dello schema S;

finché lo schema S non ha alcuna soluzione
    modifica lo schema S, togliendo un numero da una casella;
    stampa le soluzioni dello schema S;

finché lo schema S ha più di una soluzione
    scegli tra queste una soluzione, sia essa S';
    modifica lo schema S, aggiungendo in una delle caselle vuote
    il numero che sta nella corrispondente casella di S'; (*)
    stampa le soluzioni dello schema S;
```

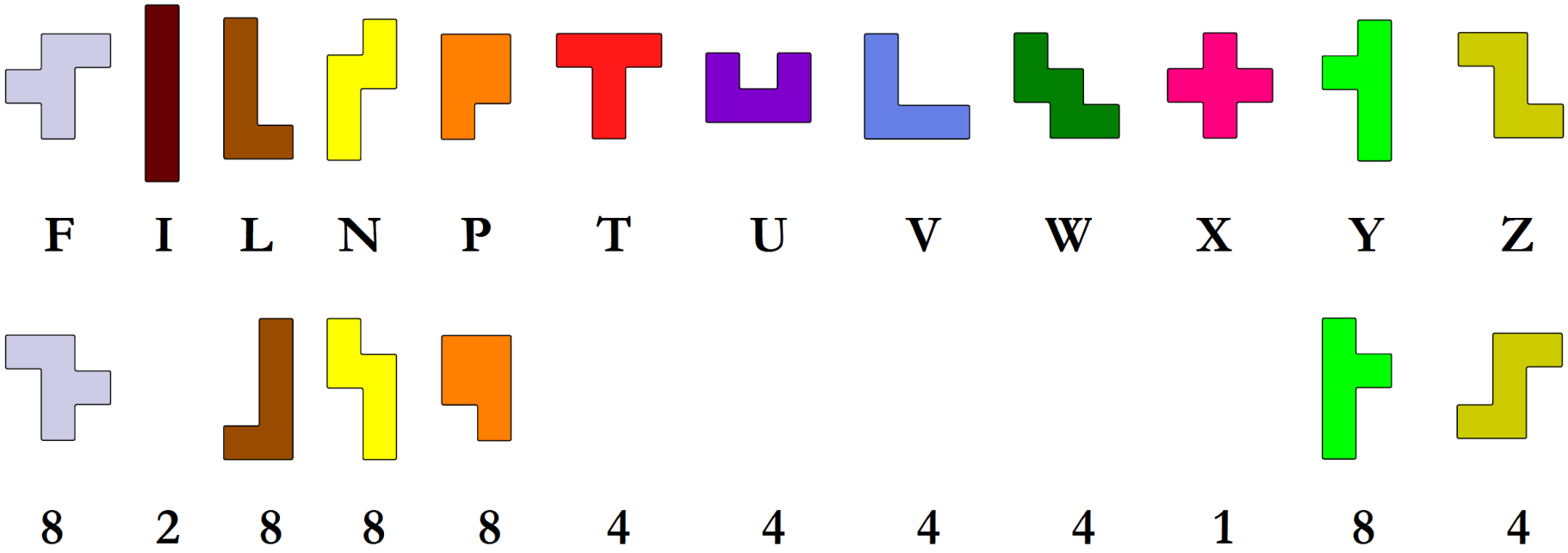
(*) Se S' fosse la sola soluzione che ha tale numero in quella casella, allora al passo successivo si otterrebbe lo schema desiderato!
Ovviamente, anziché "stampare" le soluzioni è sufficiente calcolarne il numero e, quando sono più di una, ricordarne una soltanto!

Un puzzle con i dodici pentamini e un tetramino … ma bicolori! Quante mai saranno le soluzioni?

Henry Ernest Dudeney, “74. The broken chessboard”, in “The Canterbury Puzzles”, London: William Heinemann (1907), pp. 90-92, 174-175



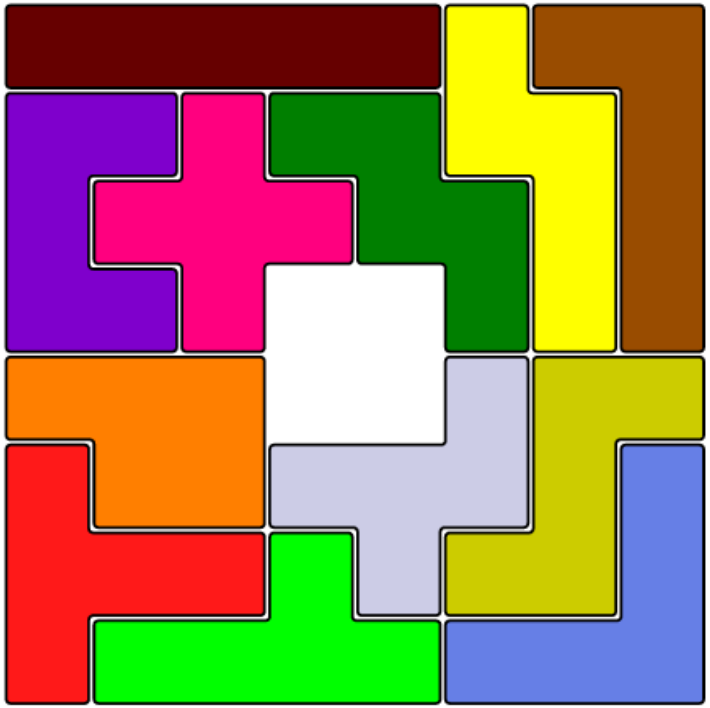
I dodici pentamini e le loro 63 diverse posizioni



Sei pentamini devono essere colorati su entrambe le facce!

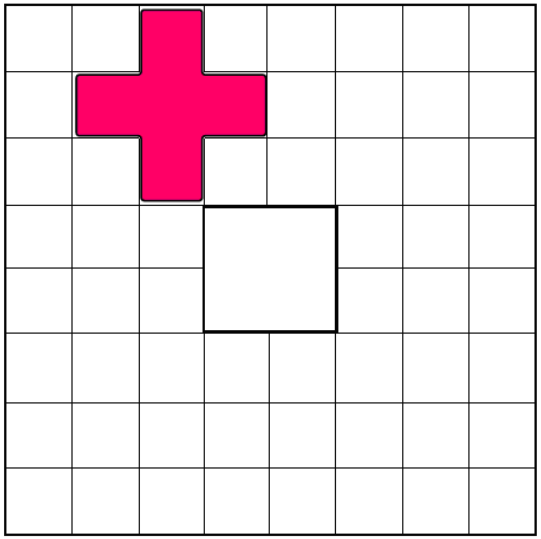
Puzzle risolto da uno dei primi programmi con *backtracking*

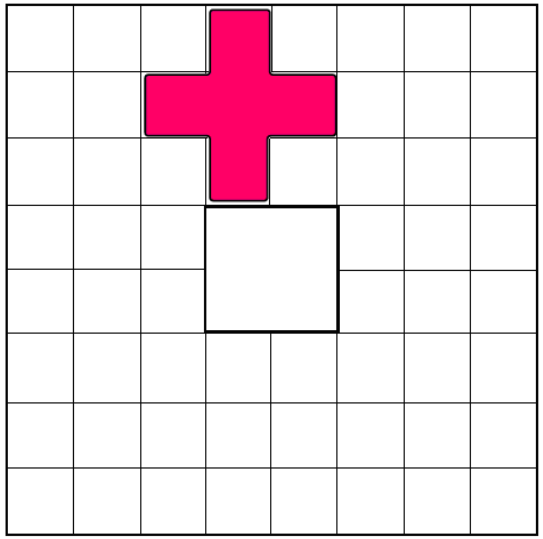
Dana S. Scott, “Programming a combinatorial puzzle”, Technical Report No. 1 (1958), New Jersey: Department of Electrical Engineering, Princeton University
(Il programma fu realizzato con l’aiuto di Hale F. Trotter.)

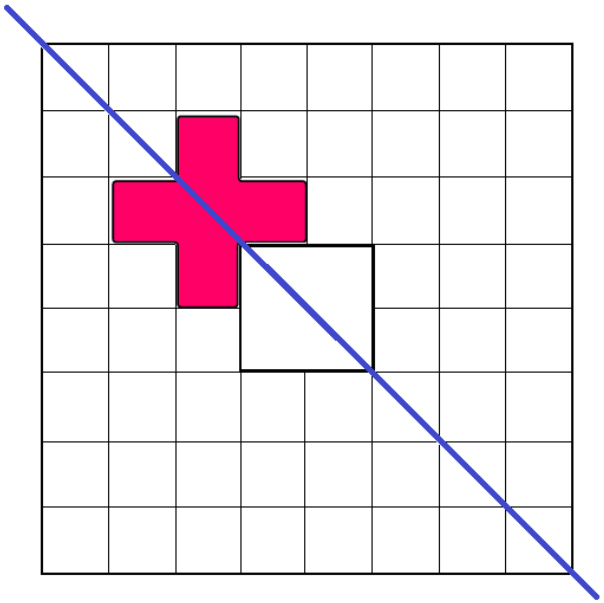


Le soluzioni diverse sono 65, trovate dal programma di Scott e Trotter in circa 3.5 ore:

19 con X centrato in (2, 3);
20 con X centrato in (2, 4);
26 con X centrato in (3, 3) e P non ribaltato (per evitare di contare due volte le soluzioni simmetriche rispetto alla direzione NW-SE).

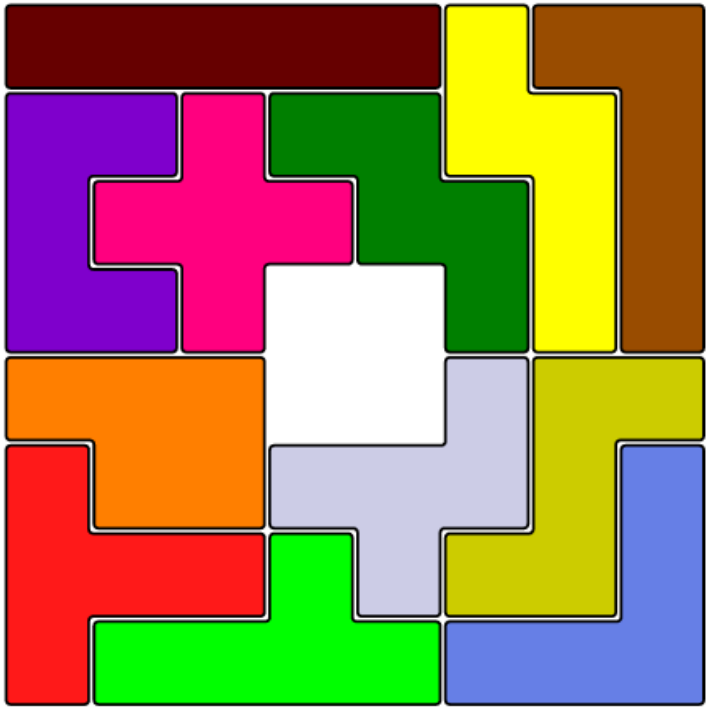






Per ciascuno di questi tre casi:

- ognuno degli altri 11 pezzi deve essere collocato una e una sola volta,
- ognuna delle 55 caselle rimaste libere deve essere occupata da uno e uno solo di essi.



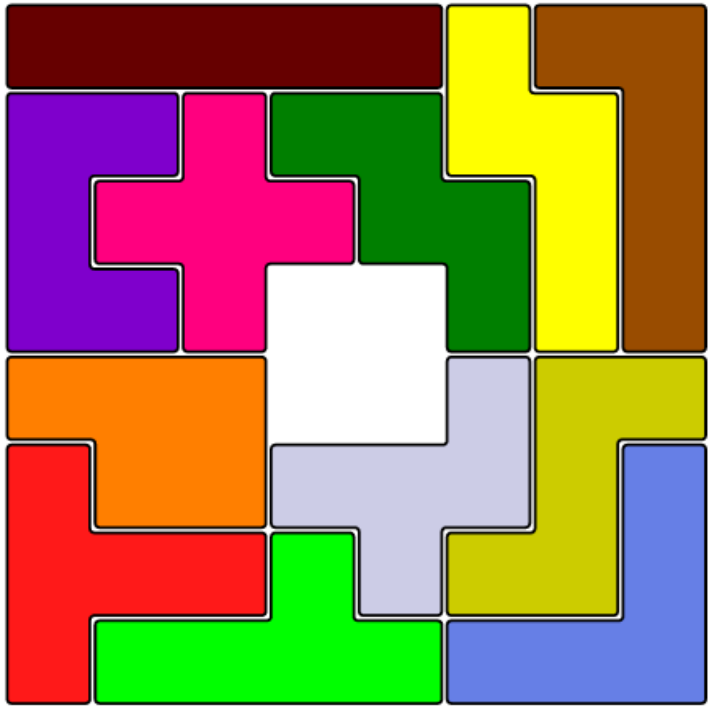
- Tre distinte istanze del problema di “esatta copertura” di un insieme

- Struttura dei dati:
una matrice di bit con 66 colonne:

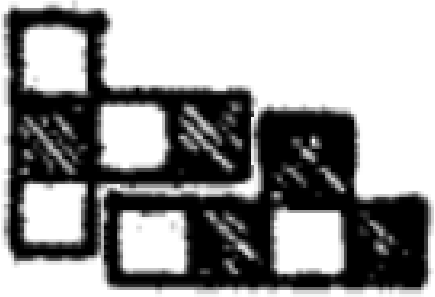
  11 i pezzi (collocato X) e
  55 le caselle rimanenti;

una riga (con esattamente sei 1) per ogni collocazione ammissibile di ciascuno degli 11 pezzi sulla scacchiera “bucata”

- Sia ciascun pezzo, sia ciascuna casella devono essere presi una e una sola volta, e dunque ad ogni sottoinsieme costituito da 11 righe, che presenti esattamente un 1 in ciascuna colonna, corrisponde una soluzione.



Se qui si aggiunge il tetramino quadrato al centro, NON si ottiene una soluzione del puzzle di Dudeney, anche potendo ribaltare i pezzi della scacchiera; infatti …



D’altronde, Dudeney NON richiede che il tetramino quadrato sia al centro − e nella sua soluzione non lo è!

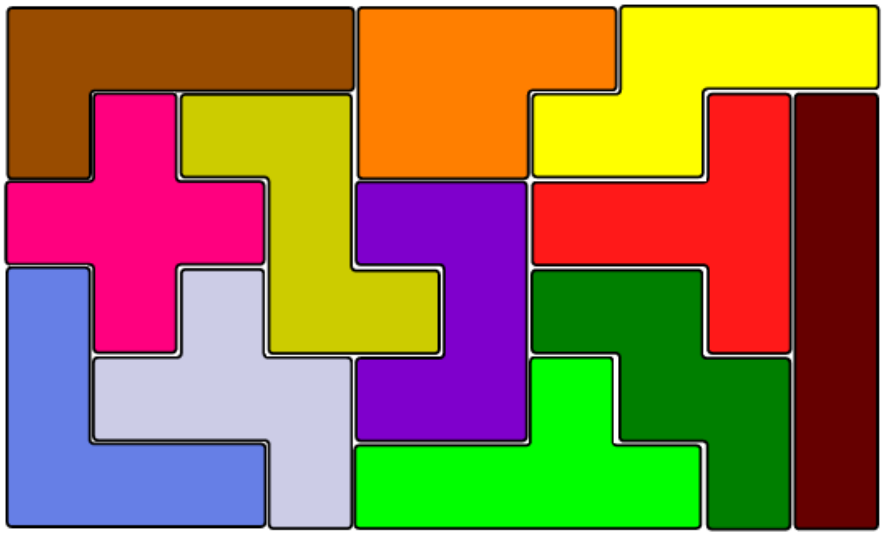
Generalizzando, il problema con i dodici pentamini in tinta unita e il tetramino quadrato in una *qualsiasi* collocazione ammette 16146 soluzioni diverse nel quadrato 8 x 8;
si veda: Donald E. Knuth, “Dancing Links”, 2000
http://www-cs-faculty.stanford.edu/~knuth/papers/dancing-color.ps.gz

Altri puzzle “onerosi”: comporre rettangoli con i 12 pentamini

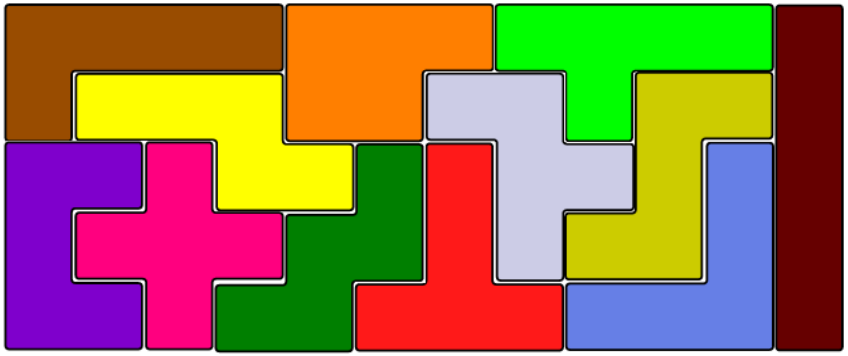
Rettangolo 6 x 10

Colin B. Haselgrove, Jenifer Haselgrove, “A computer program for pentominoes”, Eureka 23, 2 (1960), Cambridge, England: The Archimedeans, pp. 16-18

John G. Fletcher, “A program to solve the pentomino problem by the recursive use of macros”, Communications of the ACM 8 (1965), pp. 621-623
http://www.cs.virginia.edu/~skg5n/fletcher.pdf

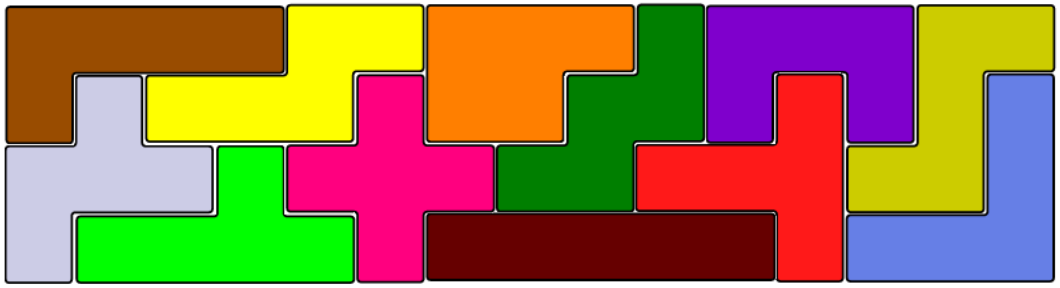


2339 diverse soluzioni, trovate dal programma di Fletcher in circa 10 minuti!



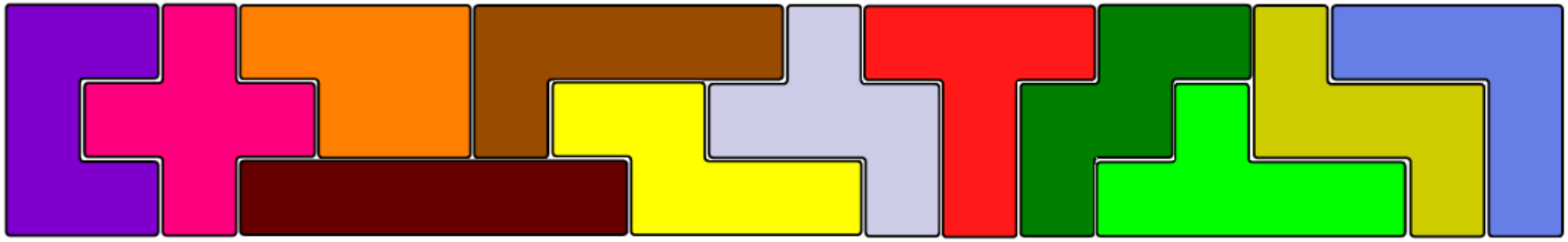
Rettangolo 5 x 12

1010 soluzioni



Rettangolo 4 x 15

368 soluzioni

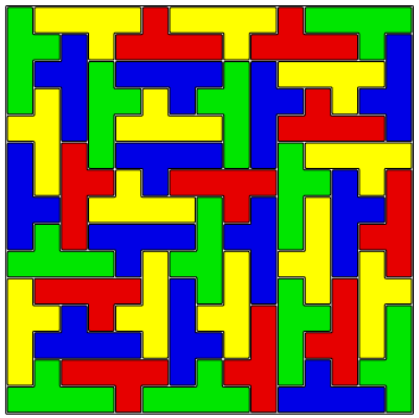


Rettangolo 3 x 20

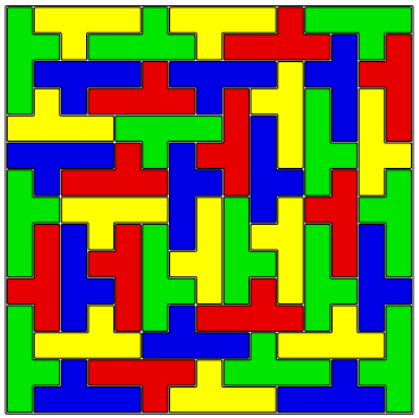
Soltanto 2 soluzioni: qual è l’altra?

Comporre un quadrato con 45 pentamini Y

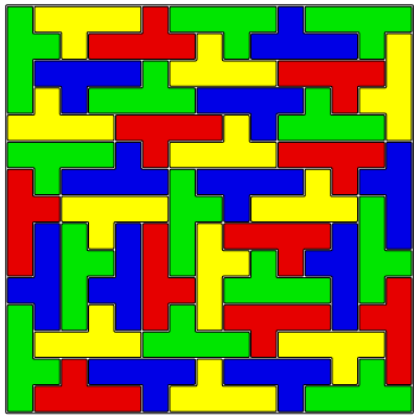
Una singola soluzione fu trovata da Jenifer Haselgrove: “Packing a square with Y-pentominoes”, Journal of Recreational Mathematics 7 (1974), p. 229.
La lista di tutte le 212 soluzioni è di Donald E. Knuth (“Dancing Links”, 2000)
http://www-cs-faculty.stanford.edu/~knuth/papers/dancing-color.ps.gz



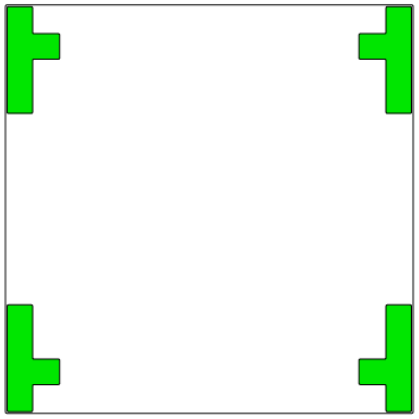
92 solutions, 14,352,556 nodes, 1,764,631,796 updates



100 solutions, 10,258,180 nodes, 1,318,478,396 updates

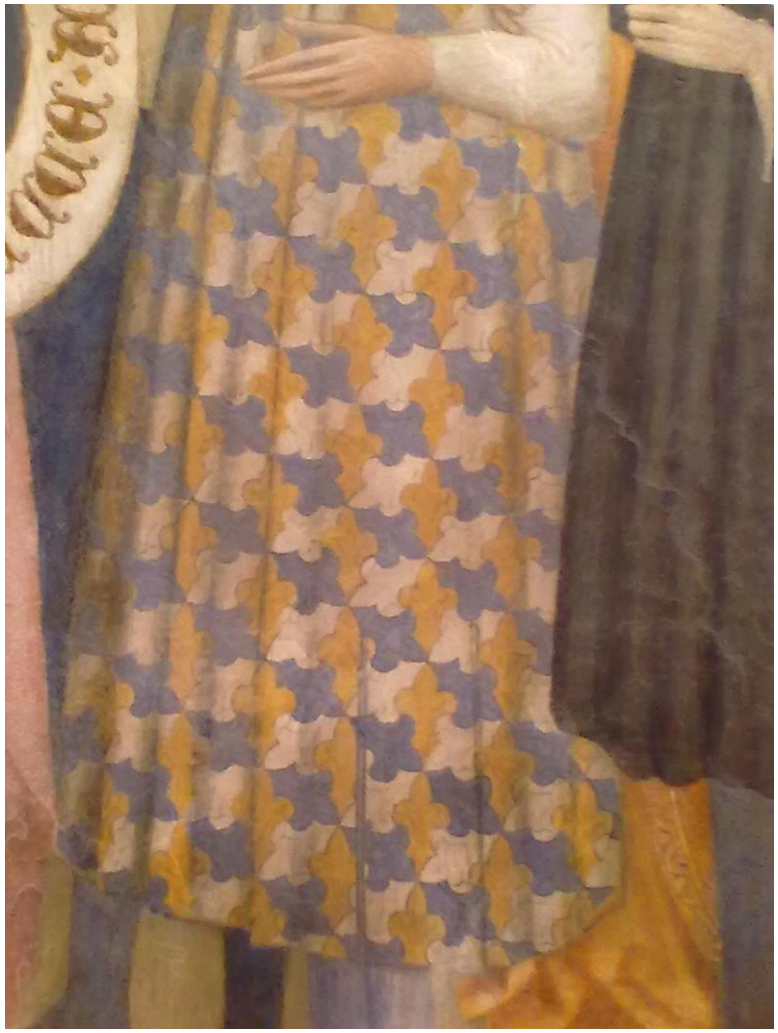


20 solutions, 6,375,335 nodes, 806,699,079 updates



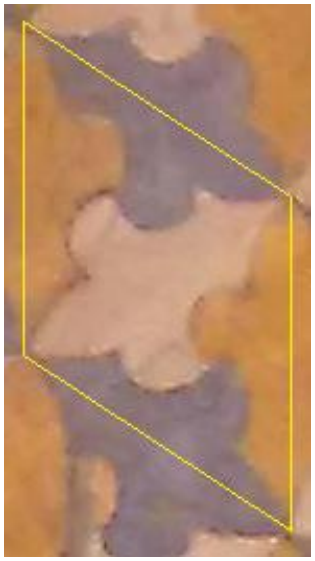
0 solutions, 1,234,485 nodes, 162,017,125 updates

Con ognuno dei 12 pentamini si può realizzare una tassellatura periodica del piano …



Tassellatura periodica:
si ripete lungo due direzioni indipendenti

→ si può individuare un parallelogrammo periodico



Uscita dal labirinto (se c'è): la prima o una migliore

```
9 10
##########
#.x..#.#.#
##.#.#.#.#
#..#...#.#
#.####...#
..#..###.#
###.##...#
#....#....
##########
```

```
##########
#.xoo#.#.#
##.#o#.#.#
#..#ooo#.#
#.####ooo#
..#..###o#
###.##..o#
#....#..oo
##########
```

E-S-N-W

13 passi

```
##########
#.xoo#.#.#
##.#o#.#.#
#..#ooo#.#
#.####ooo#
..#..###o#
###.##ooo#
#....#oooo
##########
```

E-W-S-N

17 passi

```
##########
#.x..#.#.#
##o#.#.#.#
#oo#...#.#
#o####...#
oo#..###.#
###.##...#
#....#....
##########
```

S-E-N-W

6 passi

Cambiando l'ordine delle quattro direzioni, la soluzione trovata può essere diversa.

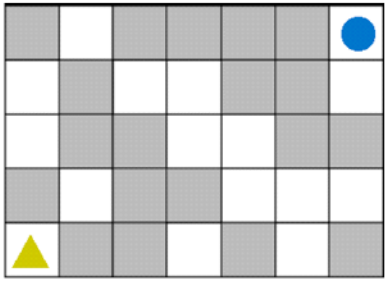
Versione ottima: si provano tutte per ottenere una delle più brevi.

Vedi codifiche in C++ nella cartella programmi_Cpp > labirinto

Tante variazioni sul tema del labirinto …

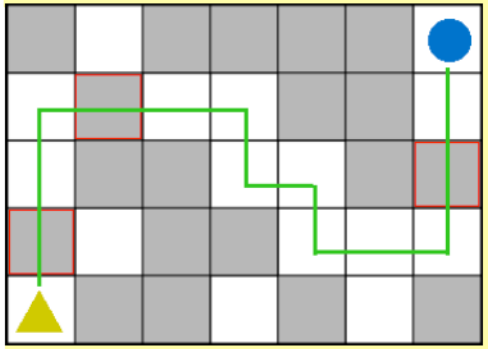
Erase walls (Slovacchia, 2017)

The maze consists of empty fields (white squares) and walls (gray squares).
We can move from one empty field to the adjacent empty field in the horizontal or vertical direction (not diagonally).



Select (by clicking) **as few walls as possible** which should be demolish in order to enable moving from the bottom left corner to the top right corner of the maze.

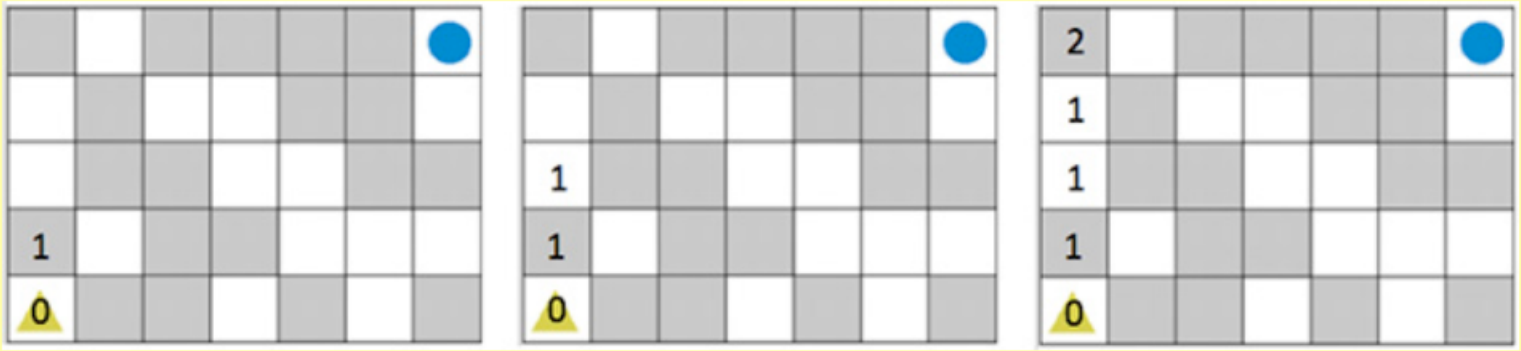
The minimum number of walls that needs to be demolished is 3.



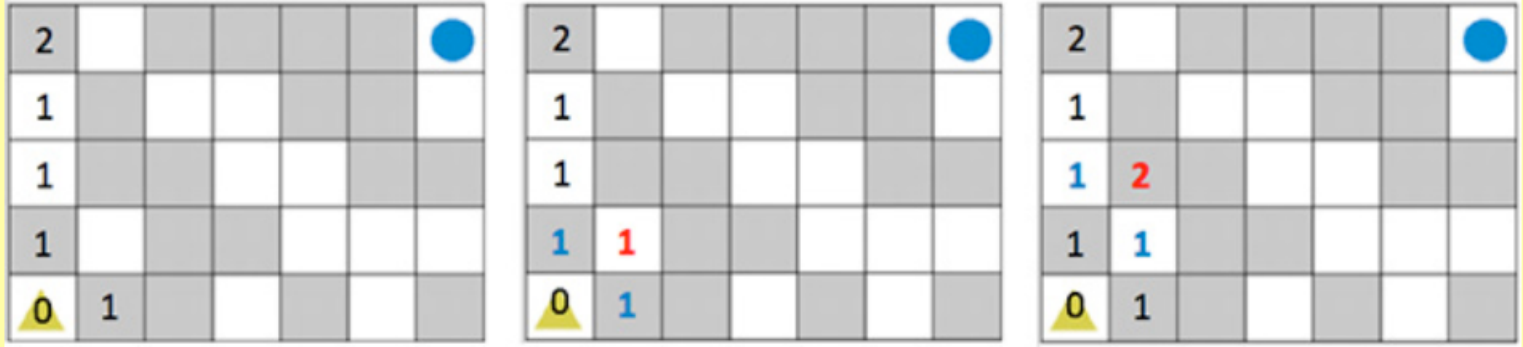
A systematic way to solve this problem is by marking each individual cell with 'the number of walls that needs to be demolished in order to reach it'. We can start with the bottom left cell, followed by the other cells in the first column.

Since the first cell (bottom left) is the starting position, we can mark it with 0. Next, we look at the second cell (the cell immediately on top of the first cell), followed by the next cell, and so on until all the cells in the first column are marked.

In each step, if the cell is a wall, we increase the number from the previous cell by one and mark it, otherwise we mark it with the same number.



Then we look at the next column. This time, we need to look at both the cell underneath it and the cell to the left of it. We select the minimum number from those two cells, and if the cell is a wall, we increase that number by one and mark it, otherwise we mark it with the same number.



Take note that after marking the cells, it is important to check the cells again from the opposite direction (in this case, from top to bottom, and from right to left), and overwrite the mark with a lower number if it is possible. Sometimes an overwritten cell will affect other adjacent cells as well, so it is important to check several times until no mark needs to be changed further. In the following pictures, the cells that have been overwritten are highlighted in yellow.

Finally, after all the cells have been marked, we can see that the final cell (top right cell) is marked as 3. This means that to reach the top right corner, a total of 3 walls have to be demolished.

Searching for a path in a maze is a known informatics problem. This task uses ideas from other similar tasks, but adds an additional requirement which is to demolish as few walls as possible.

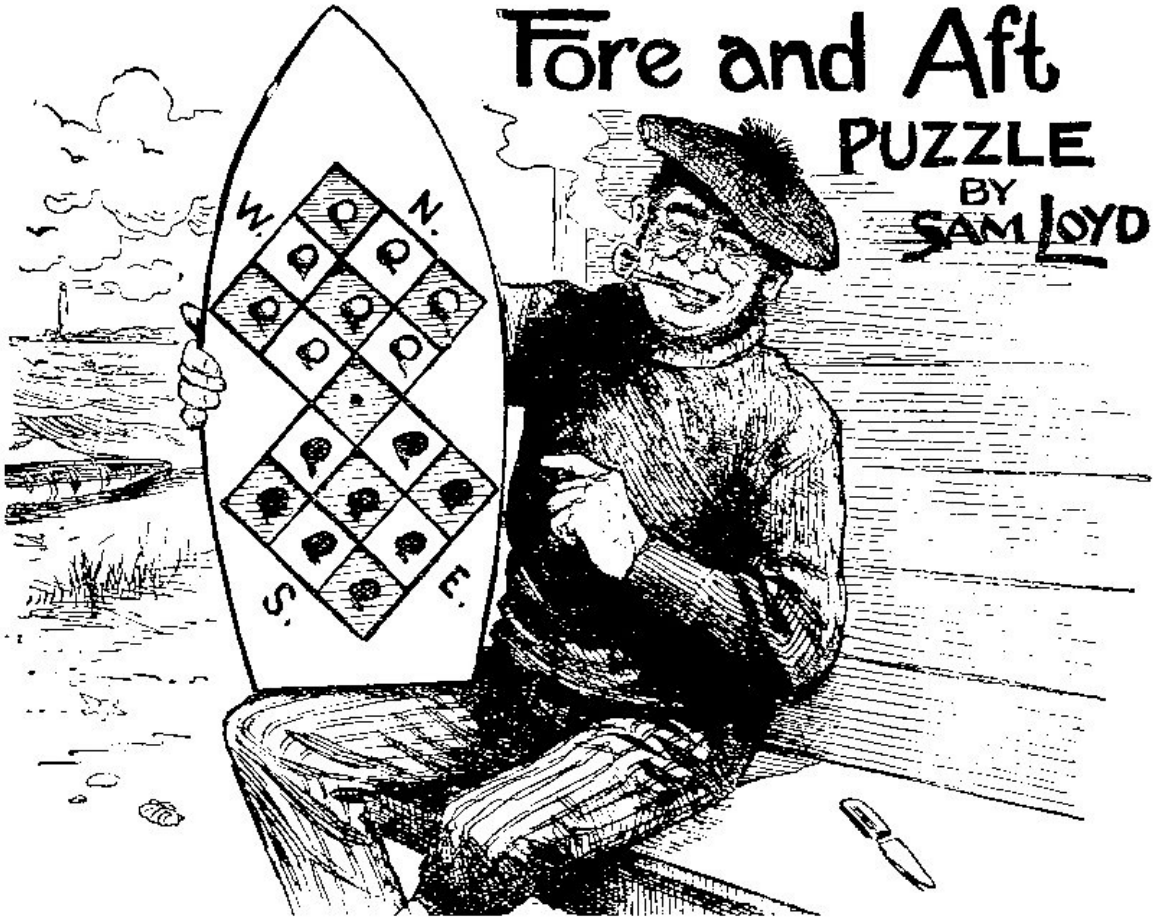
Answering this task systematically requires algorithmic thinking by scanning the cells one by one from the starting position. Marking the all the cells produce an array that contains valuable information: the number of walls needed to be demolished to reach each cell. In computer science, an array is a data structure consisting of a collection of elements, such as values or variables.

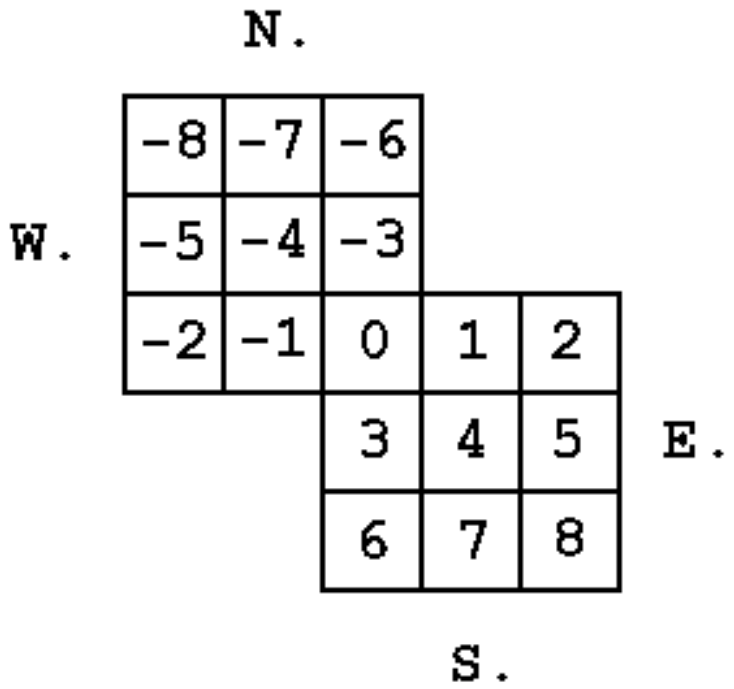
❑ Puzzle un po' più complicati …

- Una soluzione, se c'è, consiste nella sequenza di mosse che ha portato a uno stato finale, magari già noto, ma dal quale non può essere dedotta.
- Talvolta **non** si rischia di cadere in cicli (ad es. se le pedine non possono retrocedere o se ne è tolta una ad ogni mossa)
- … ma **spesso** invece occorre mantenere una "lista" degli stati toccati lungo il percorso fatto …
- Volendo poi trovare tutte le soluzioni più brevi, oltre a forzare il *backtracking*, occorre mantenere un "insieme di soluzioni" (ciascuna soluzione è una "lista di stati"):
  - quando è trovata una soluzione di pari lunghezza, è aggiunta alle altre;
  - se è trovata una soluzione più breve, rimpiazza tutte le altre.

A prua e a poppa

W. W. Rouse Ball (1892), Sam Loyd (1914)





Soluzione in 46 mosse, particolarmente elegante per la sua simmetria, trovata da H. E. Dudeney nel 1898:

1 **-1** -2 **0** 3 **-3** -6 **0** **6** 7 **1** **-1** 0 **2** 5 **3** **-3** -4 -1 **-7** -8 **-2** **0**
2 **8** 7 **1** 4 3 **-3** **-5** -2 **0** 1 **-1** **-7** -6 **0** **6** 3 **-3** 0 **2** 1 **-1** 0

In 46 mosse, numero minimo, ve ne sono altre 2475 che iniziano con 1.
(Le 75 soluzioni più lunghe che iniziano con 1 richiedono 58 mosse.)

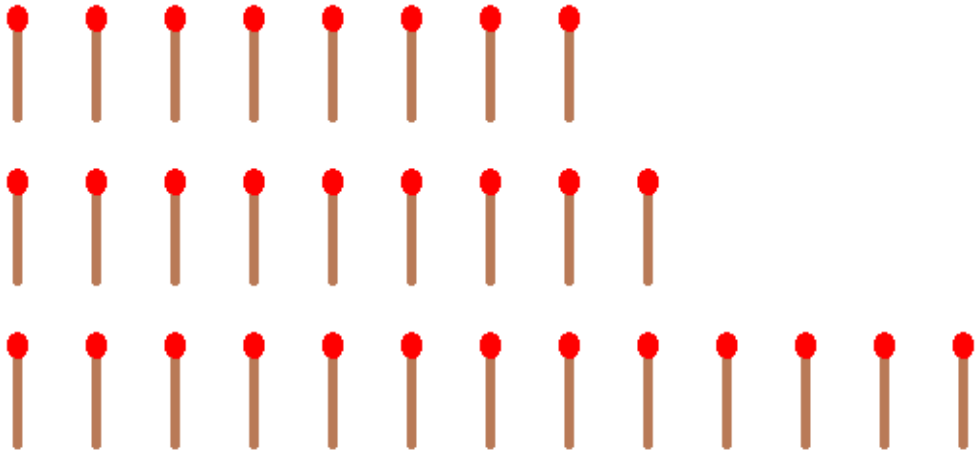
❑ *Giochi per due …*

- Come “risolvere” un gioco tra due avversari (sufficientemente semplice, ad esempio il tris o una sua variante, vedi M. Gardner)
- Come realizzare un gioco tra due avversari fermando l’analisi a una certa profondità
- Come determinare quale dei due avversari ha la strategia vincente in un gioco combinatorio imparziale (di vario genere: NIM, il gioco di Euclide, Babylone …)
- Giochi di altro tipo, ad esempio *MasterMind*: come programmare il ruolo di solutore …

NIM (Charles L. Bouton, 1901)

• A turno, ciascuno dei due giocatori prende quanti fiammiferi vuole (almeno uno) da una sola delle file in cui sono ripartiti.
• Vince chi prende per ultimo.

Scriviamo in binario il numero di fiammiferi in ciascuna fila e calcoliamo l'*even parity bit* per ogni colonna di cifre binarie:



```
1  0  0  0
1  0  0  1
1  1  0  1
-----------
1  1  0  0
```

Per vincere, bisogna ridurre a 0 tutti i bit di parità …

Qui chi gioca ha la possibilità di vincere; tre le mosse “giuste”:

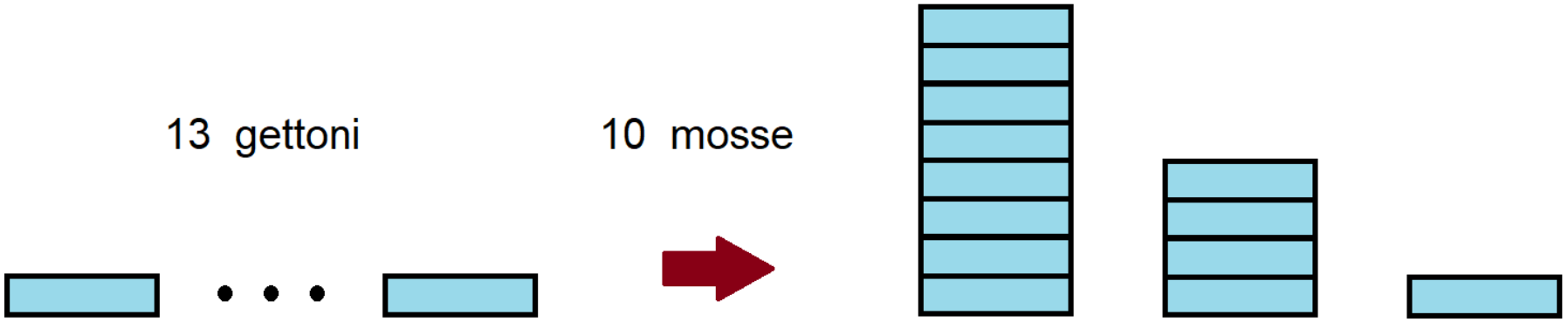
- togliere 4 fiammiferi dalla prima fila, oppure
- togliere 4 fiammiferi dalla seconda fila, oppure
- togliere 12 fiammiferi dalla terza fila.

```
0 1 0 0        1 0 0 0        1 0 0 0
1 0 0 1        0 1 0 1        1 0 0 1
1 1 0 1        1 1 0 1        0 0 0 1
---------      ---------      ---------
0 0 0 0        0 0 0 0        0 0 0 0
```

Se nella prima fila vi fossero stati 6 fiammiferi anziché 8, allora una soltanto sarebbe stata la mossa giusta: quale?

Attenzione ai giochi …

- infiniti: Pong Hau K'i, Picaria …
- in cui lo stesso giocatore può sempre vincere: Kayles …
- in cui entrambi i giocatori sono "dummy": Babylone-one …



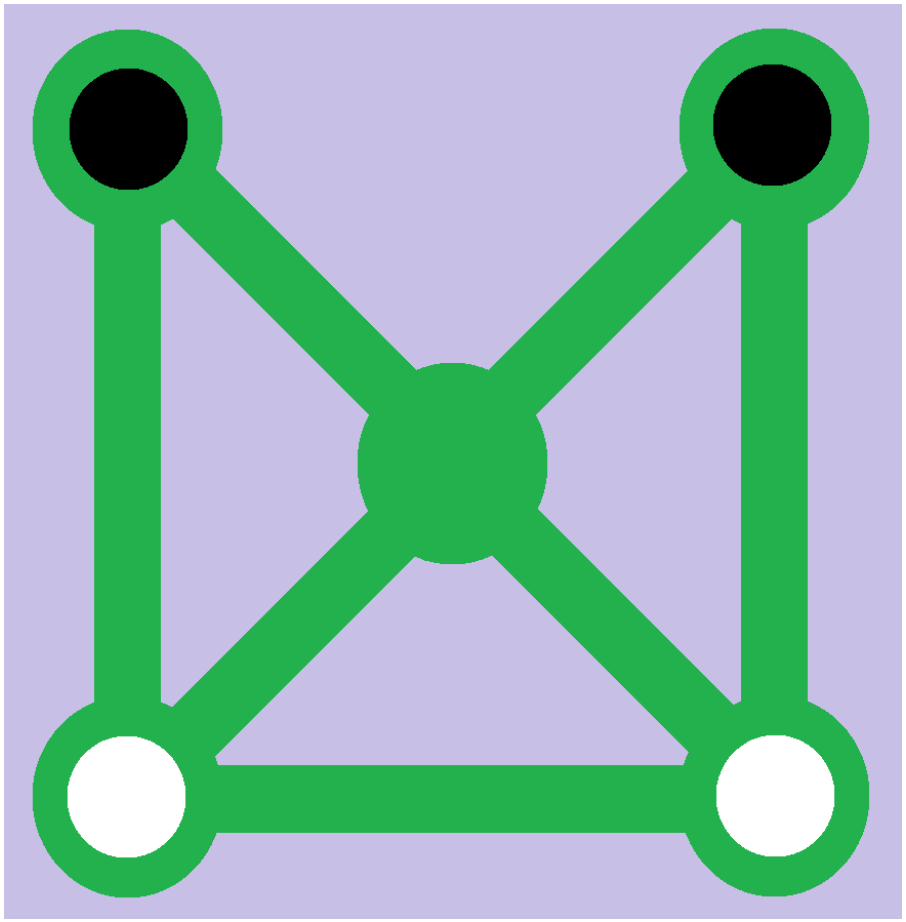
$13 = 8 + 4 + 1 = 1101_2$ → vince il secondo (idem con 12)

→ tolto l'ultimo bit, se i bit 1 sono pari vince il secondo, altrimenti vince il primo, comunque muovano: "ininfluenti"!

→ sequenza del vincitore non periodica (Prouhet-Thue-Morse)

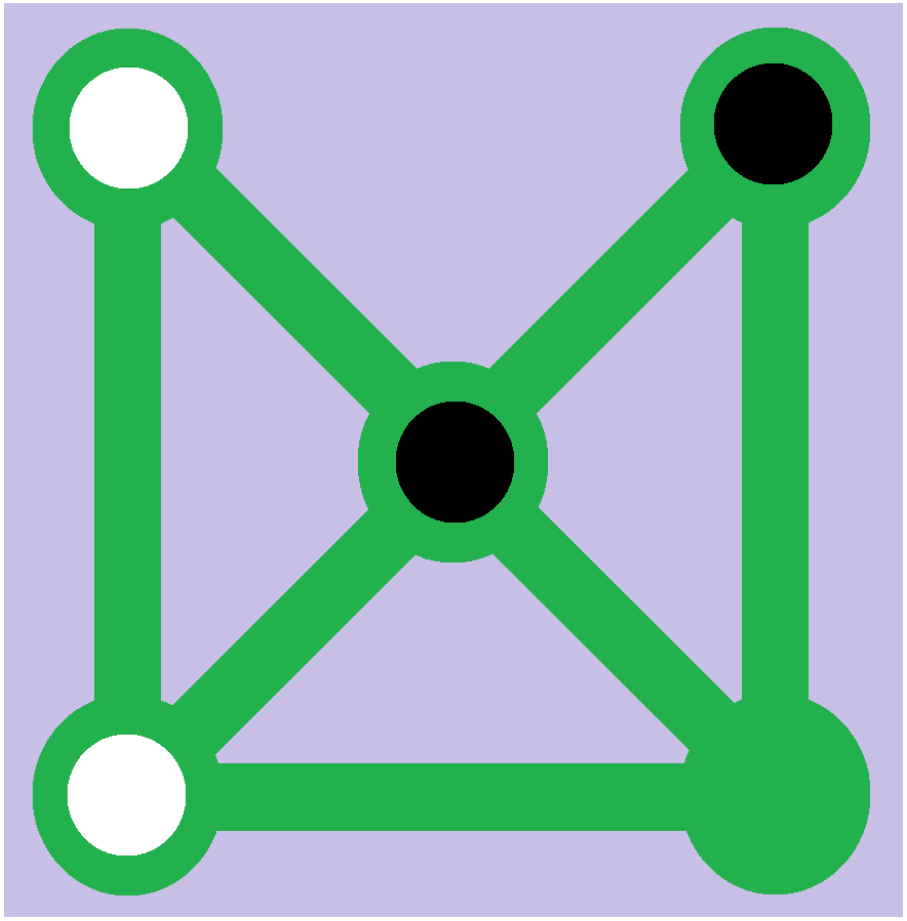
… da non confondere col "gioco di Grundy" (1939).

Un gioco infinito: Pong Hau K'i



- A turno, ciascun giocatore sposta una delle pedine del proprio colore lungo un lato, sino al vertice libero.
- Vince chi riesce a impedire all'avversario qualsiasi mossa.

Supponiamo che inizi il Bianco e che si giunga allo stato qui sotto raffigurato, con mossa al Nero: allora il Nero vince!



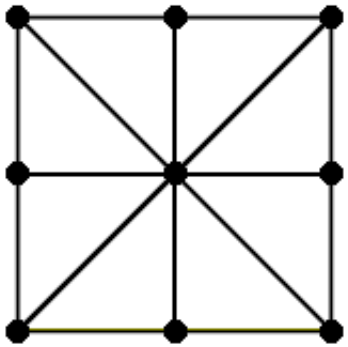
- Tuttavia, se nessuno commette errori, la partita non termina:

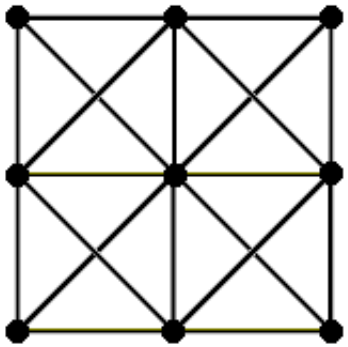
chiunque inizi, nessuno dei due giocatori ha modo di forzare il blocco dell'avversario!

Sono tantissimi i giochi con pedine su tavolieri…

Qualche esempio, dove i due giocatori collocano a turno le proprie pedine, prima di muoverle, e lo scopo è fare un tris:

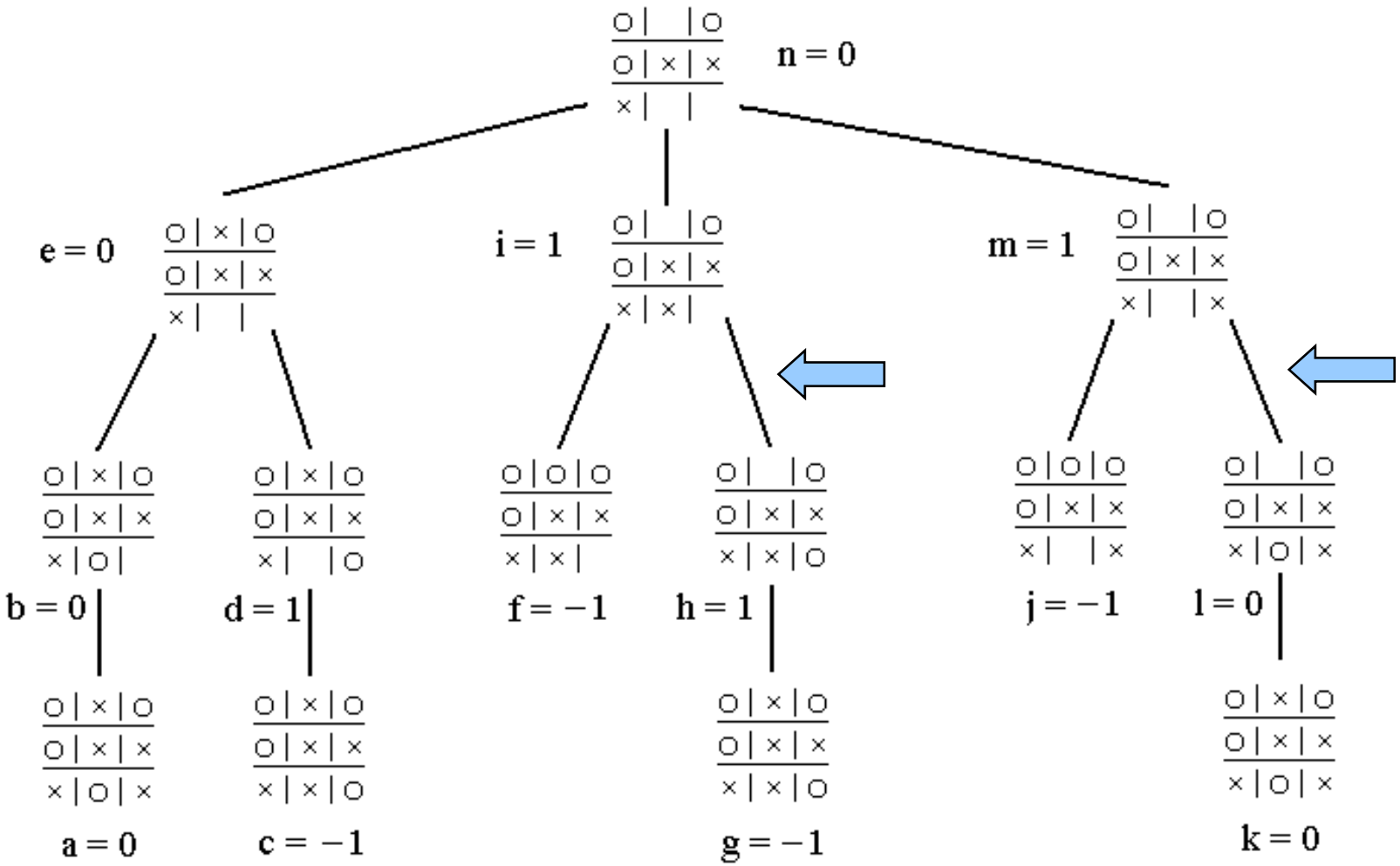
- Tapatan / Achi con 3 / 4 pedine a testa sul tavoliere a sin.
  Ha una strategia vincente il primo giocatore.
  Achi presenta interessanti varianti…





- Picaria con 3 pedine a testa sul tavoliere a destra
  Nessuno dei due giocatori può forzare la vittoria:
  è un gioco infinito!

Giochi a due, determinati, a somma 0: il tris



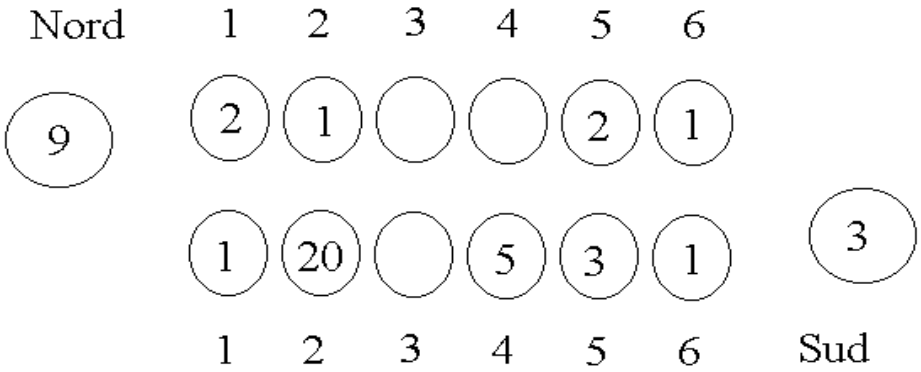
Alcuni giochi risolti (a favore della parità)

- Tris: 765 stati (modulo simmetrie) di cui 138 finali,
  26830 partite (differenti sequenze di stati)
- Tela classica: circa 8 miliardi di stati (1995)
- Awari: circa 900 miliardi di stati (2002)
- Dama 8×8: circa $5 \cdot 10^{20}$ stati (2007)

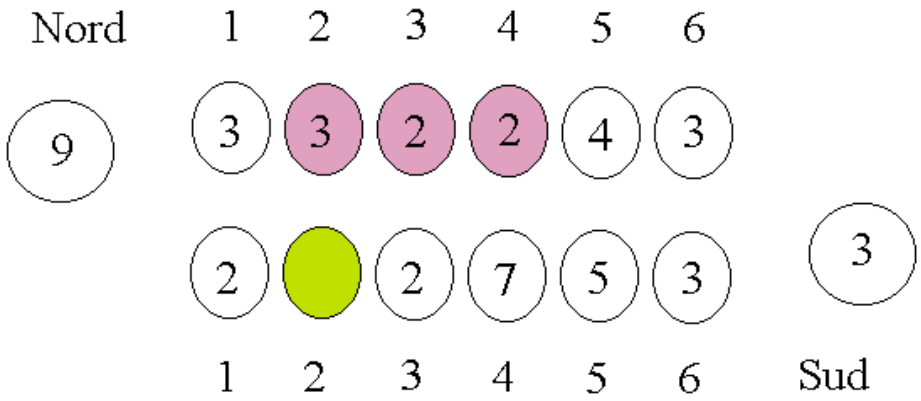
Alcuni giochi che *mai* saranno risolti

- Scacchi: numero di stati stimato con 47 cifre decimali
- Shōgi: numero di stati stimato con 71 cifre decimali
- Go 19×19: numero di stati stimato con 171 cifre decimali

Awari



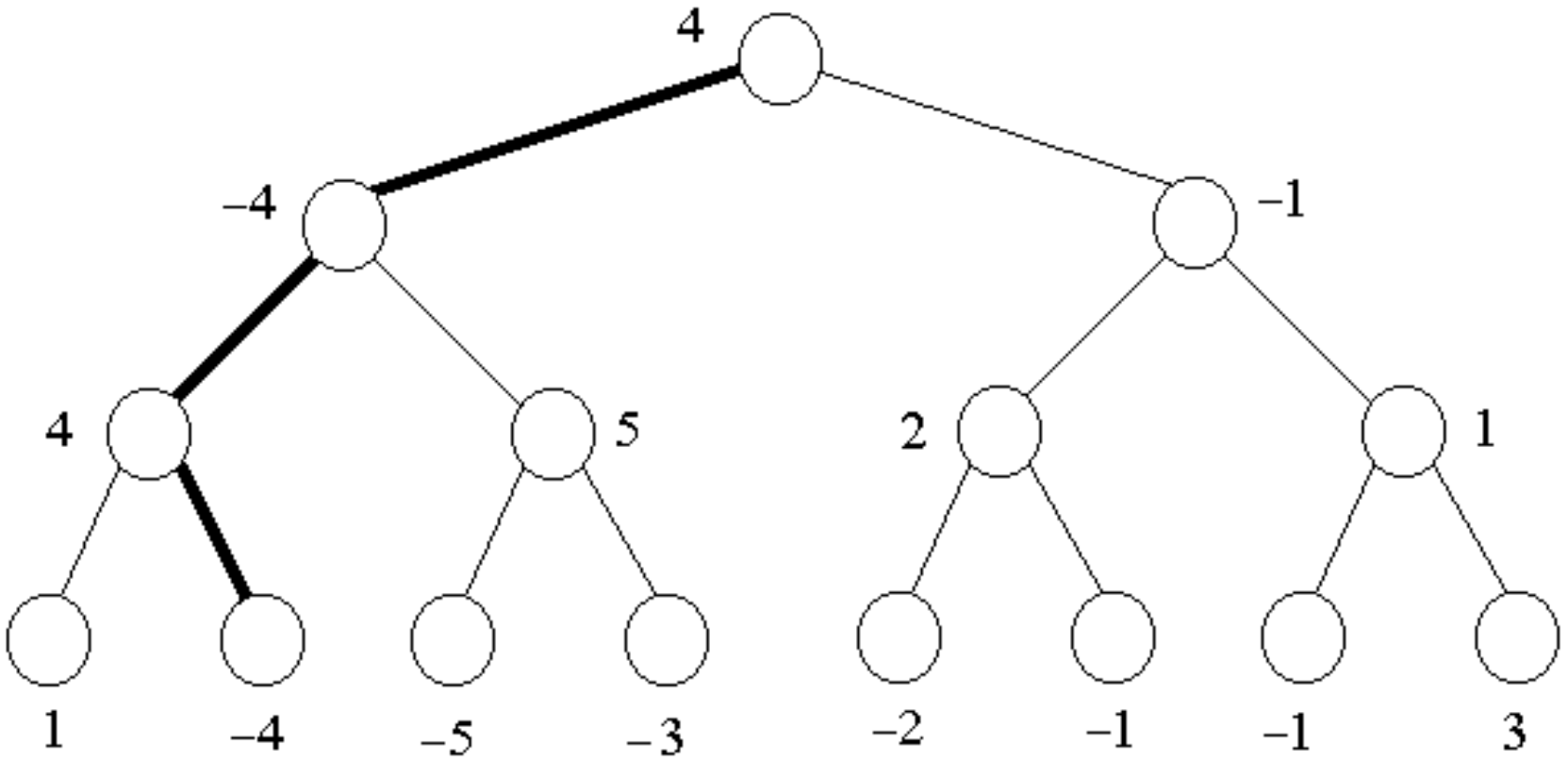
Stato del gioco dopo Sud 2:



… e Sud guadagna 7 semi.

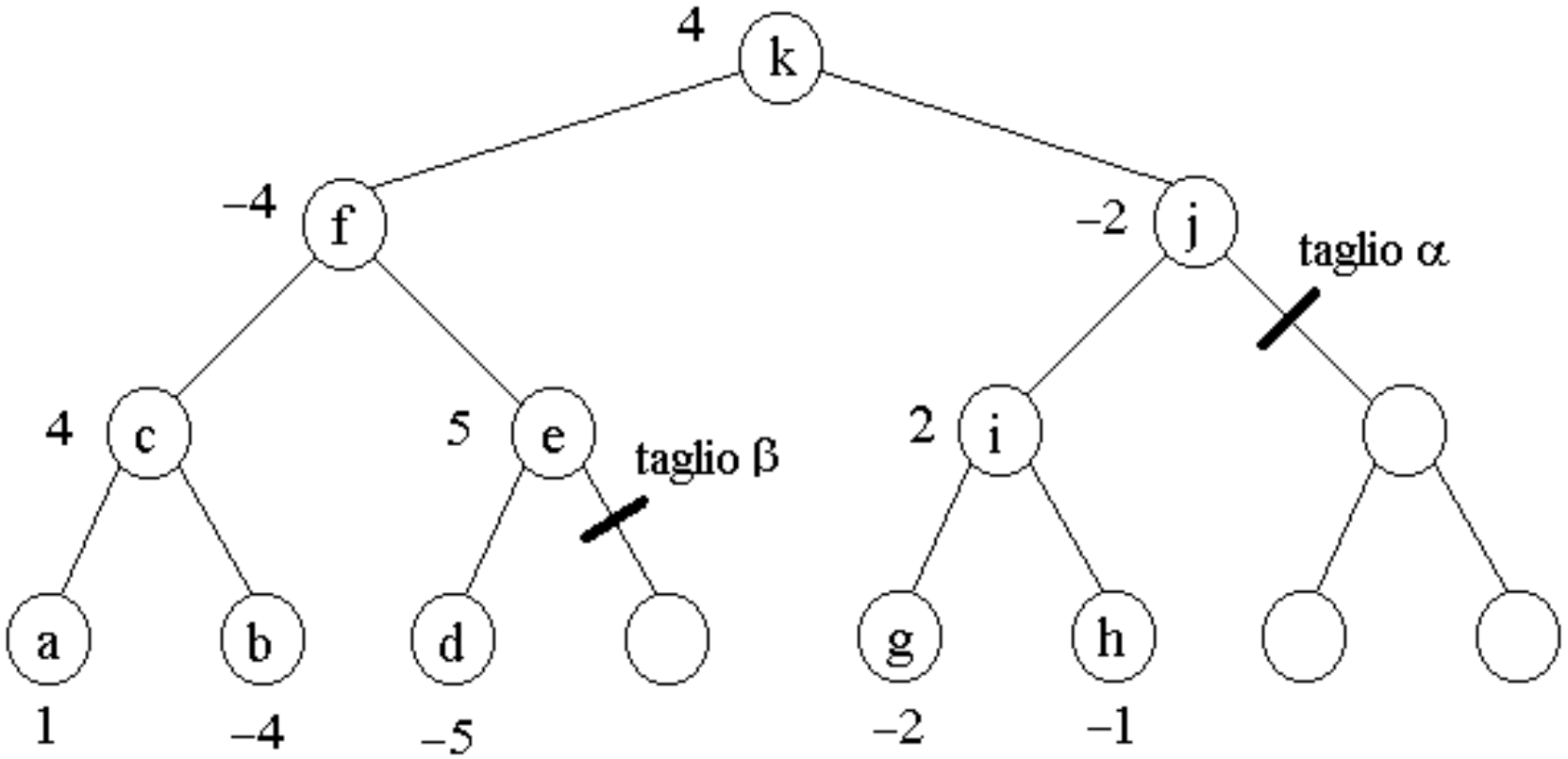
Risolto in senso forte: per ognuno degli stati (essenzialmente diversi) è stata trovata la lista delle mosse “giuste” da fare!

Come realizzare un programma che giochi bene



- Minimizzare la massima perdita possibile (su un orizzonte)
- Ricordare le varianti principali!

Potatura alpha-beta (J. McCarthy, 1956)



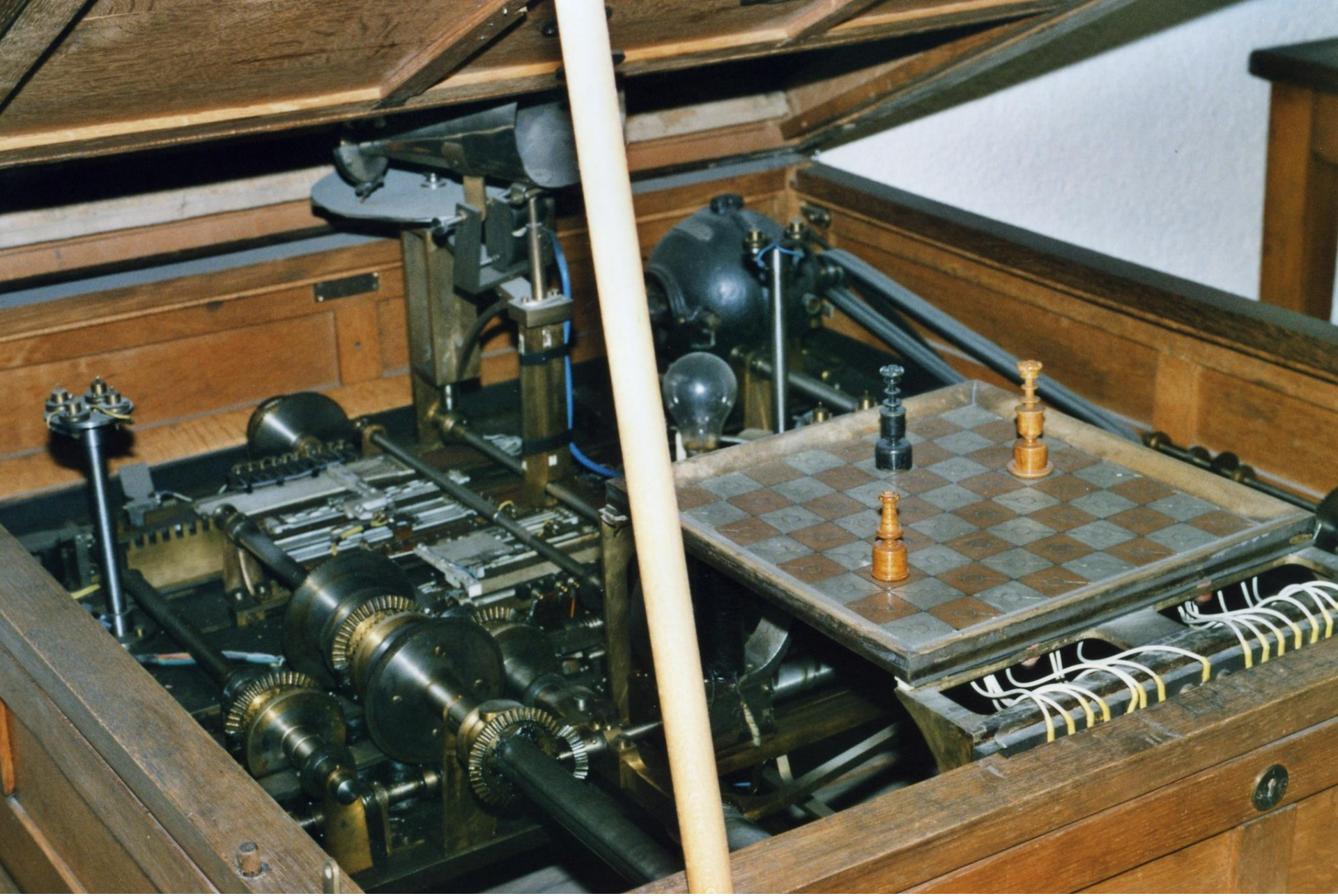
- Applicata agli scacchi da Newell, Simon e Shaw (CIT, 1958).
  Awari a profondità 8: risparmio di tempo > 50%

Miglioramenti

- Minimal Window Principal Variation Search:
  T. A. Marsland e M. Campbell (1982-85)
- Nega-Scout: A. Reinefeld (1983-89)

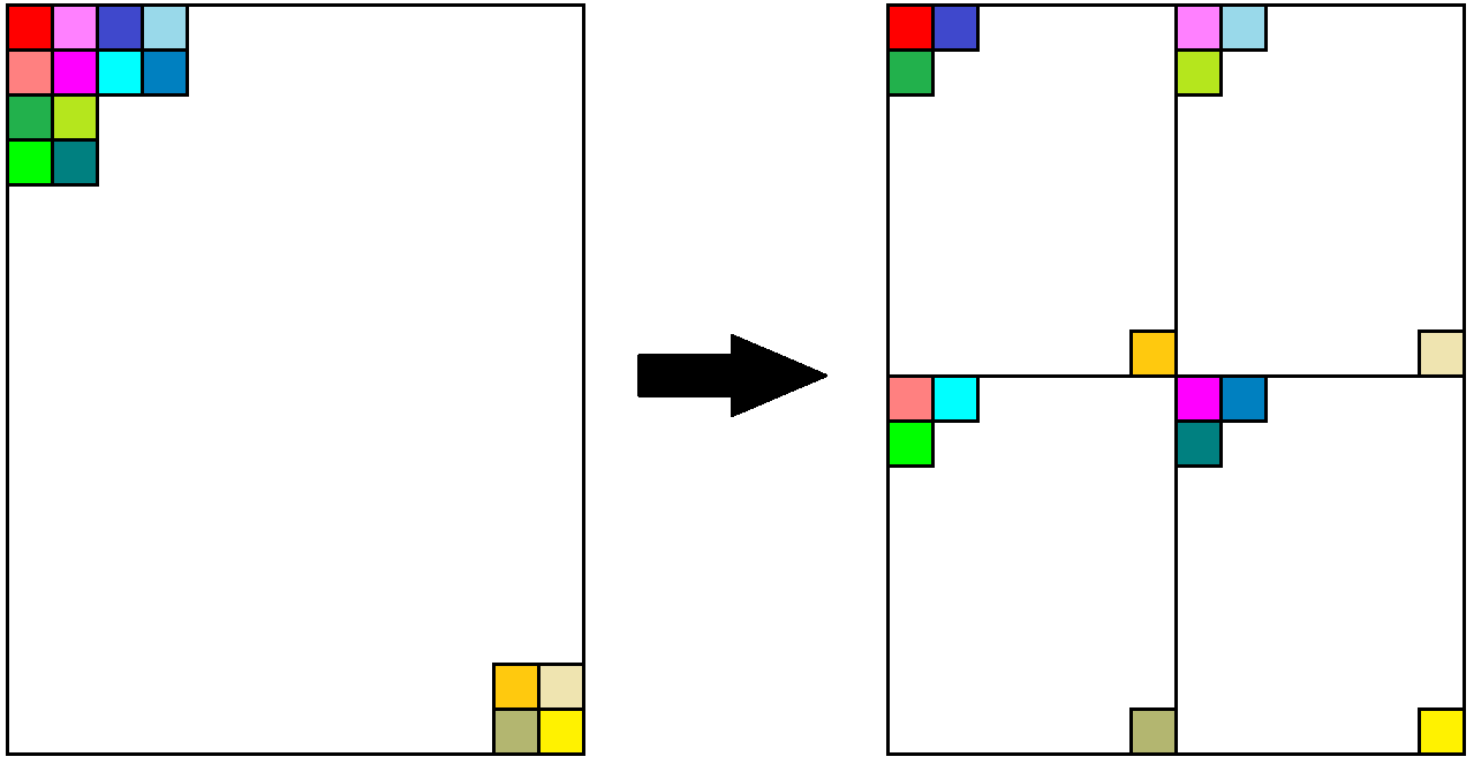
Awari a profondità 14: risparmio di tempo > 15% vs alpha-beta

- L'efficienza aumenta notevolmente se le mosse lecite sono almeno una ventina e sono ordinate in una lista best-first.
- Iterative deepening: ordinamento best-first a ciascun livello di profondità, prima di passare al livello successivo.
- Transposition table: per evitare di analizzare più volte uno stesso stato del gioco.



L. Torres y Quevedo, El segundo ajedrecista, 1920 (Politecnico di Madrid, foto dell'autore)

Photomaton (J.-P. Delahaye e P. Mathieu, 1997)



Se il formato dell'immagine è $2m \times 2n$ pixel,
e p_1 = il più piccolo intero tale che $2m - 1$ divide $2^{p_1} - 1$
e p_2 = il più piccolo intero tale che $2n - 1$ divide $2^{p_2} - 1$
allora l'immagine iniziale riappare dopo mcm(p_1, p_2) passi.

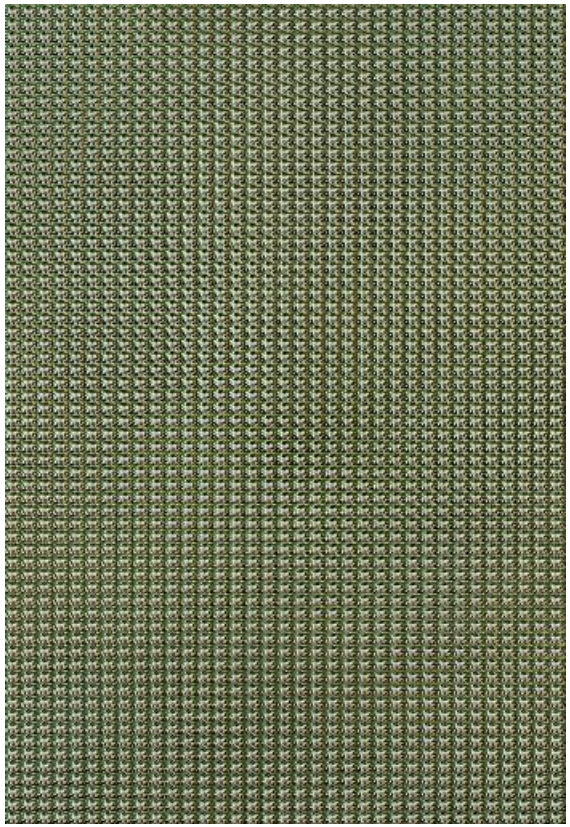
- L’immagine del gatto è di formato 350 × 512 pixel.
- Il più piccolo intero p_1 tale che 349 divide $2^{p_1} - 1$ è $348 = 2^2 * 3 * 29$;
- il più piccolo intero p_2 tale che 511 divide $2^{p_2} - 1$ è $9 = 3^2$.
- Dunque, l’immagine del gatto riapparirà dopo 1044 passi…

dopo un passo…

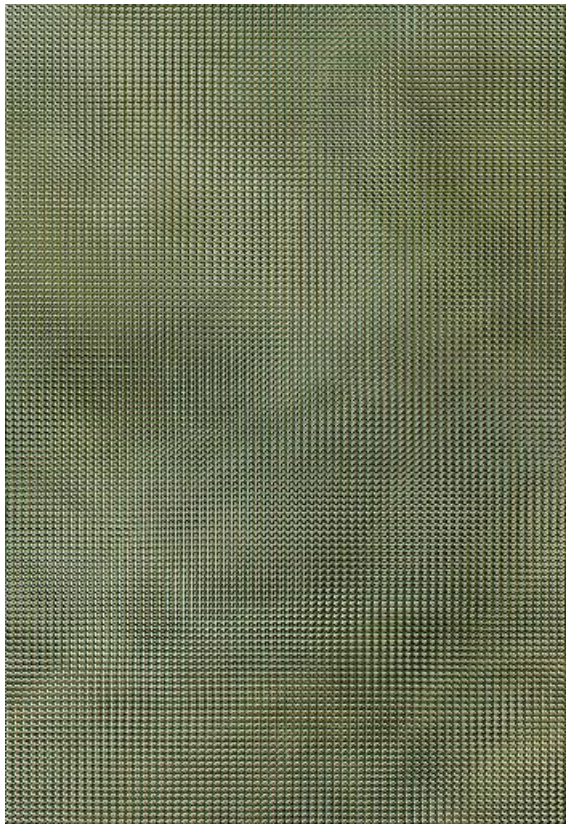
dopo due…

Tuttavia, alcuni passi intermedi danno curiosi risultati; ad esempio:

- dopo 488 passi l'effetto è quello di vedere 5 (!) x 4 gatti "ribaltati"…



dopo 33 passi…



dopo 34…

dopo 488…

- dopo 522 passi (la metà di 1044: ma sarà sempre così?) l'immagine originale riappare perfettamente ribaltata lungo l'asse verticale;
- infine, dall'immagine del passo 1043, dove la testa del gatto s'intravede ingrandita, si ritorna in un solo passo all'immagine nitida di partenza!

dopo 522 passi…

dopo 1043…

dopo 1044

Lorenzo Repetto

Dai giochi agli algoritmi

Un'introduzione non convenzionale all'informatica

Edizioni Kangourou Italia