# “Problemi, soluzioni, codifiche” – Didascalie per le diapositive

**2.** Il metodo descritto (e provato) da Euclide per il calcolo del massimo comun divisore è ritenuto il primo significativo esempio di algoritmo. Altrettanto famoso è il “crivello” ideato da Eratostene di Cirene per eliminare via via i numeri non primi dalla successione dei naturali.

**3.** Spesso si legge, o si sente dire, che Ada Byron è stata la “prima programmatrice” della storia; in realtà, ella aiutò Charles Babbage (e ne sostenne le idee e le iniziative) dopo che questi, nel 1834, concepì il progetto della “macchina analitica”, peraltro mai costruita: un enorme calcolatore meccanico, mosso da un motore a vapore, comandato da sequenze di istruzioni codificate su schede perforate, comprese diramazioni condizionate per poter eventualmente ripetere o saltare una sotto-sequenza di istruzioni (a seconda del segno del risultato dell’ultima operazione), e dunque impiegabile per calcolare qualsiasi “formula analitica”… Ada ebbe comunque la capacità di vedere questa macchina in modo ancor più astratto: per elaborare informazioni, non soltanto per eseguire calcoli!
Il “programma” attribuito ad Ada, il cui lavoro si svolse a stretto contatto con Babbage, è frutto del tentativo di formalizzare in una procedura il calcolo dei numeri di Bernoulli (che, in effetti, è una successione di razionali) servendosi di una formula iterativa: lo commenteremo, traducendolo in ambiente Maple, in due diverse modalità, sintatticamente quasi identiche ma con esiti assai differenti.

**5.** Le tre equazioni, mutuamente esclusive, definiscono una funzione totale, dalle coppie di naturali ai naturali, che non può essere espressa con gli usuali cicli **`for`** (ove è previsto un numero di iterazioni sì variabile in funzione dei dati, ma precalcolato).

**6.** Per qual motivo, all’inizio del nostro discorso, abbiamo detto “prima del 1945”? Era ancora il 1941, quando l’ingenere civile tedesco Konrad Zuse completò la realizzazione del primo computer digitale, pienamente operativo, tutto a relé elettromagnetici e controllato da programma (esterno, a sola lettura, perforato su una pellicola da film)… Ma durante gli ultimi anni del secondo conflitto mondiale, rifugiatosi in un piccolo villaggio sulle Alpi bavaresi, Zuse inventò e sviluppò un complesso linguaggio di programmazione, che chiamò *Plankalkül*. Questo fu il primo linguaggio di alto livello ad essere concepito, ma rimase sulla carta: all’epoca non poteva darsi altrimenti; comunque anticipò – seppur prematuramente – alcune importanti idee anche per l’organizzazione e l’elaborazione delle informazioni. Zuse specificò il suo *Plankalkül* non tanto formalmente, quanto piuttosto attraverso numerosissimi esempi di applicazione, tutti riportati in un manoscritto di trecento fitte pagine…

**8.** Qui ho trascritto (e corretto) uno di questi esempi: si tratta dell’ordinamento per inserimento, a scansione diretta, particolarmente adatto nelle situazioni in cui la sequenza da ordinare è in realtà già quasi ordinata, ossia presenta relativamente pochi valori fuori posto. Il manoscritto che lo codifica precede di circa un anno le lezioni di John Mauchly e la pubblicazione delle relative dispense, nel 1946, presso la Moore School of Electrical Engineering, sulle tecniche di ordinamento (sia interno, sia esterno) comprensive di questo metodo.

**9.** I *flow-chart* furono introdotti da Herman H. Goldstine e John von Neumann nel 1947, corredati di appositi blocchi contenenti asserzioni automaticamente soddisfatte ogniqualvolta il controllo vi passa, e ancor oggi simili diagrammi sono usati…

**10.** … forse in maniera un po’ “intricata”, a dispetto delle raccomandazioni di mezzo secolo fa sui buoni principî di programmazione!

**11.** Ecco due aspetti di fondamentale importanza in programmazione, che ne scoprono le radici logico-matematiche, … già evidenziati negli anni ’40 ma a lungo trascurati dagli sviluppatori di software!

**12.** Il “pensiero computazionale” può essere sviluppato a partire dalla scuola primaria…

**13.** … con l'ausilio di vari strumenti; uno dei più validi, a mio parere, è XLogo, un recente dialetto (sviluppato presso il Politecnico Federale di Zurigo) del famoso linguaggio LOGO, che nacque nel 1967 ad opera principalmente di Seymour Papert.

**16.** Può risultare un po' difficoltoso scrivere i comandi giusti per ricollocare la tartaruga allo scopo di disegnare una figura che si ripete: bisogna infatti fare attenzione alla posizione e all'orientamento che essa assume quando termina il disegno di ciascuna figura.

**17.** Si può mettere in risalto, con semplici esempi, anche la struttura gerarchica di un programma…

**23.** Un altro ambiente adatto ai primi approcci è Scratch, basato sulla composizione di blocchi (che si istanziano sullo schermo e poi si collegano tra loro o si innestano l'uno nell'altro, trascinandoli con il mouse), sviluppato presso il MIT all'inizio di questo secolo.

**27.** In Python, in una lista di $n$ elementi (ovvero di lunghezza $n$), agli elementi è associato un indice da 0 a $n - 1$. Una buona idea è partire dall'ultimo arrivato, risalendo man mano fino al primo…

**29.** … e pensare che questi – tutti risolvibili producendo algoritmi assai efficienti! – sono tra i problemi per certi aspetti più facili proposti alle Olimpiadi di Informatica a Squadre!

**30.** Soffermiamoci piuttosto su alcuni significativi esempi di programmi (e di comandi) in ambiente Maple, che possiamo trovare nella cartella programmi_Maple …
Il problema dell'isomorfismo tra grafi, così come quello della fattorizzazione, sta in una specie di limbo, a quanto pare poco popolato: per questi problemi non sono stati trovati (almeno finora) metodi generali efficienti (tempo-polinomiali), né si è provato che siano tanto impegnativi quanto i cosiddetti "NP-ardui" (quelli che si sospettano veramente intrattabili quando la loro dimensione supera un certo limite). Nello stesso limbo, fino al 2002, si trovava un altro problema famoso: quello della primalità.

**39.** Con "algoritmo efficiente" gli informatici intendono un procedimento che, per essere eseguito, richiede un tempo, *al più, polinomiale*, cioè limitato superiormente da un polinomio, nella lunghezza dei dati in ingresso codificati in bit. (Usualmente, poi, s'intende che tale polinomio non debba avere un grado troppo elevato!)
*Minimum Spanning Tree*, ossia *albero ricoprente di costo minimo*: è un sottografo *connesso minimale* (togliendo uno qualsiasi dei suoi archi, non è più vero che da ciascun nodo si può raggiungere ogni altro nodo) o, equivalentemente, *aciclico massimale* (aggiungendovi un arco si crea inevitabilmente un ciclo, cioè un cammino chiuso), comprendente tutti i nodi del grafo (non orientato e pesato), e tale che la somma dei costi degli archi che lo compongono sia la più piccola possibile.
Il più antico procedimento risolutivo per questo problema, e forse anche il più facile da realizzare, è l'*algoritmo di Borůvka*, ideato da Otakar Borůvka nel 1926 per la costruzione di una rete elettrica in Moravia: la rete deve giungere in tutti i punti prestabiliti (i nodi del grafo) senza creare percorsi chiusi; i costi sono le distanze tra due punti (per tutte le coppie di punti). L'*algoritmo di Kruskal* (Joseph B. Kruskal, 1956) è un altrettanto semplice esempio di algoritmo *greedy* (goloso, ingordo): ad ogni passo, si collegano i due nodi più vicini, facendo soltanto attenzione a che non si creino cicli. Infine, l'*algoritmo di Prim* (Robert C. Prim, 1957, ma già pubblicato da Vojtěch Jarník, in un articolo in lingua ceca, nel 1930) ispirò l'*algoritmo di Dijkstra* (Edsger W. Dijkstra, 1959), ancora *greedy*, per trovare i cammini di costo minimo da un nodo a tutti gli altri (più in generale, in un grafo orientato).
Visitare un grafo (anche orientato) a partire da un certo nodo significa, in pratica, costruire un albero che tocchi tutti i nodi raggiungibili da quello di partenza (prescindendo dagli eventuali costi)…
Con "problema difficile" gli informatici intendono, sostanzialmente, un problema (in generale) per la cui esatta risoluzione non si conosce alcun algoritmo efficiente: o non si conosce perché si sa che non esiste proprio (come accade per i problemi intrinsecamente esponenziali, ad esempio le *torri di Hanoi*), o non si conosce e non si sa neppure se ne esista uno. Per i problemi qui menzionati vale quest'ultima accezione, e si tratta di una notevole questione aperta dell'informatica teorica.

*Problema del ciclo (o circuito, se il grafo è orientato) hamiltoniano*: si deve trovare, se c'è, un cammino chiuso che includa tutti i nodi, una e una sola volta ciascuno, e se il grafo è pesato (ossia con un costo associato a ciascun arco) si richiede anche che tale cammino chiuso sia di costo minimo (*Travelling-Salesman Problem*, ossia *problema del commesso viaggiatore*).
*Bin-Packing*, ossia *problema dell'imballaggio*: disponendo di contenitori (*bin*) tutti uguali, ciascuno dei quali sopporta un certo peso massimo *P*, e data una lista di *n* numeri ($p_1$, …, $p_n$) che rappresentano i pesi di *n* oggetti, si tratta di ripartire tutti questi *n* oggetti nel minor numero di contenitori, in modo tale, ovviamente, che la somma dei pesi in ciascuno di essi non superi *P*. (Tutti questi numeri sono intesi essere interi positivi, con ognuno degli *n* pesi minore o uguale a *P*.)
*Multiprocessor Scheduling*: eseguire un insieme di processi (di ognuno dei quali si conosce la durata) su un certo numero di processori (identici) in parallelo, facendo sì che il tempo richiesto per eseguirli tutti sia minimo; gli eventuali vincoli di precedenza sono espressi da un grafo orientato aciclico.
Partendo da quesiti proposti in diverse gare di informatica per studenti della scuola secondaria, illustreremo i problemi "di sottoinsieme" che qui sono stati citati…

**40.** … ma prima vorrei discutere un problema di *Minimum Spanning Tree*, tratto dalla gara finale della prima edizione del *Kangourou dell'Informatica* (maggio 2009), con qualche ritocco… Sembra facile codificare l'idea. Tuttavia l'idea giusta prevede di ripartire inizialmente i vertici in altrettanti insiemi; due vertici si potranno collegare se appartengono a insiemi diversi (altrimenti si formerebbe un ciclo), e, se si collegano, i rispettivi insiemi dovranno essere uniti…

**44.** Ecco un esempio molto semplice di grafo: è una mappa, dove accanto ai vari tratti è riportato il tempo di percorrenza (che qui può variare a seconda del verso). Più precisamente, in questo caso si parla di grafo orientato e pesato. Ha senso chiedersi quale sia il cammino più breve tra due dati punti o tra tutte le coppie di punti, ammesso che la destinazione sia raggiungibile.
Di solito, i grafi che rappresentano mappe sono "planari"; tuttavia, i procedimenti ideati per questo tipo di problema sono efficienti in ogni caso.

**45.** Fu un grande informatico, l'olandese Edsger W. Dijkstra, a ideare un procedimento efficiente per trovare il cammino più breve tra due specificati nodi di un grafo qualsiasi.

**53.** L'ultima frase offre spunti di riflessione… Vi sembrerà strano, ma cercare il percorso *più lungo* da un nodo a un altro in un grafo (non passando mai due volte in uno stesso nodo), oppure stabilire se c'è un percorso che passi una e una sola volta per ciascun nodo (un cosiddetto *cammino hamiltoniano*), sono problemi per i quali non si conoscono, e non si sa neppure se esistano, metodi efficienti per risolverli in generale.

**54.** Poiché ogni arco ha il "costo" di un minuto, basta una visita in ampiezza!

**56.** Per la visita in profondità (più in generale, in un grafo orientato, ossia con archi diretti) ci soccorre una delle tecniche più generali di programmazione, detta *backtracking*. Dal nodo in cui ci troviamo, ci spostiamo – seguendo un arco – sul primo nodo non ancora visitato; quando l'esplorazione da quest'ultimo nodo sarà conclusa, ci sposteremo sul secondo nodo raggiungibile non ancora visitato, e così di seguito. Quando si sono esaurite tutte le possibilità di spostamento dal nodo in cui ci troviamo, torniamo indietro di un arco, cioè ci riportiamo sul nodo dal quale siamo giunti… e se questo non è più possibile, vuol dire che il nodo in cui ci troviamo è quello di partenza e tutti i nodi, che da questo sono raggiungibili seguendo un cammino, sono stati visitati. Il *backtracking* è utilmente impiegato nella ricerca *esauriente* − vale a dire "a tappeto" − di una o di tutte le soluzioni di un problema (qui tutte, ricordando il cammino corrente…), secondo l'idea ricorrente di esplorazione in avanti e – in caso di fallimento – ritorno sui propri passi, sino a trovare una via alternativa non ancora esplorata.

**57.** *Chiusura transitiva*: se c'è un arco da *a* a *b* e c'è un arco da *b* a *c*, allora aggiungiamo (se non c'è già) un arco da *a* a *c*. Se sulla diagonale principale della nuova matrice di adiacenza ci sono tutti 0, il grafo originario è *aciclico*.

**60.** Esistono problemi *non risolubili* (almeno in via automatica, persino disponendo di una quantità illimitata sia di memoria sia di tempo), e questo si sa già dagli anni ’30 del secolo scorso, prima ancora che un computer “universale” fosse costruito…
Tra quelli invece risolubili da una macchina, certi richiedono una quantità di risorse accettabile, pensiamo soprattutto al tempo: per tali problemi si conoscono procedimenti risolutivi *efficienti*, vale a dire che al crescere della dimensione del problema il tempo per risolverlo aumenta sì, ma non in modo eccessivamente drammatico (almeno entro certi limiti), come già abbiamo avuto occasione di vedere.

**61.** Per altri problemi risolubili, invece, si sa per certo che proprio non può esistere alcun metodo risolutivo efficiente: sono dunque problemi di fatto *intrattabili* al crescere della dimensione dei dati di input, poiché il tempo richiesto per risolverli diviene rapidamente proibitivo, persino per i super-computer più veloci al mondo.
Tuttavia non è stato facile trovarne di non banali: il primo, nel 1972, riguarda un problema *decisionale* (con risposta “sì” o “no” soltanto) nell’ambito della teoria dei linguaggi formali.
Il confine tra queste due classi di problemi (“trattabili” e “intrattabili”, secondo le accezioni assunte) non è affatto netto, anzi si tratta di una grande zona assai misteriosa, in cui si trovano tantissimi problemi interessanti. Due di questi (fattorizzazione, isomorfismo tra grafi) li abbiamo già citati…
In quello stesso anno 1972 furono “scoperti” parecchi problemi “difficili” (cioè assai impegnativi, anche per un potente computer, quando la loro dimensione cresce), per i quali fino ad oggi non è stato trovato alcun procedimento efficiente in generale, né si sa se esista; vale a dire che non è stato neppure dimostrato che il tempo *necessario* per risolverli cresca in modo esponenziale.

**62.** Questa è l’istanza del *problema dello zaino*, con varianti, proposta alla gara finale del *Kangourou dell’Informatica*, nel maggio del 2011.

**66.** Per comporre una certa somma, scegliamo ad ogni passo il taglio più grande possibile.
Caratterizzare i sistemi monetari, per i quali questo procedimento “goloso” porta al minimo numero di monete, è – se non sbaglio – una questione tuttora irrisolta qualora i tagli disponibili siano più di cinque… Questo problema, in realtà, è una variante proprio di *Knapsack*.

**68.** Questa tabella di 8 righe per 8 colonne fu presentata in un quesito della selezione scolastica, nell’ambito delle Olimpiadi di Informatica del novembre 2016, accompagnata dal seguente testo.
L’obiettivo è, partendo da una casella della prima riga (R1), arrivare a una casella dell’ottava riga (R8), minimizzando la somma dei valori nelle caselle dalle quali si passa. Le mosse consentite sono: un passo verso l’alto in verticale, un passo verso l’alto a sinistra, un passo verso l’alto a destra; per esempio, dalla casella (R2, C3) si può andare in (R3, C3) o in (R3, C2) o in (R3, C4).
Quanto vale la somma minima di un percorso dalla prima all’ultima riga?
Per rispondere a questa domanda occorre anche determinare un percorso ottimo! *Idem* se fosse richiesta la somma massima… Si può evitare una ricerca esauriente, a tappeto, che esamini *tutti* i percorsi possibili, da ognuna delle caselle di R1 a ognuna delle caselle di R8? [che sono $2^r + (c-2)\cdot 3^{r-1}$ in totale, dove $r$ = numero di righe e $c$ = numero di colonne, e dunque nel nostro caso ben 13378…]
La risposta, affermativa, ci viene dalla *programmazione dinamica*, una delle più generali tecniche algoritmiche insieme con il *backtracking* e la *programmazione lineare*. (Parlando di programmazione lineare, il termine “programmazione” ha poco a che fare con la stesura di codici per computer: si riferisce piuttosto alla *pianificazione* degli impieghi di certe risorse al fine di rendere massimo il profitto che ne deriva. L’etimologia di “programmazione dinamica” è simile: tale locuzione fu infatti concepita per l’ottimizzazione di processi multi-stadio, in particolare per quelli la cui evoluzione possa essere descritta da un grafo orientato privo di cicli.)
Come possiamo procedere? Partiamo dalla penultima riga (R7), e per ognuna delle sue caselle ci chiediamo: se siamo giunti qui, dove ci conviene salire? Ad esempio, se siamo giunti in (R7, C4), a prescindere dal percorso fatto e dalla somma finora accumulata, ci conviene salire verso l’alto in verticale, aggiungendo 5 al valore 6 della casella (R7, C4), mentre se dovessimo totalizzare la somma

massima ci converrebbe salire alla riga superiore verso destra, aggiungendo 9 al valore 6. Scriviamo dunque queste informazioni nella casella (R7, C4). Così facciamo per tutte le caselle di R7, e quindi scendiamo in R6. Anche qui procediamo in modo analogo per ciascuna casella della riga. Ad esempio, quando siamo in (R6, C5), che ha valore 7, ci conviene salire a sinistra, aggiungendo 11; dovendo invece totalizzare la somma massima, dovremmo salire a destra, aggiungendo 25 (calcolato allo stadio precedente, come somma di (R7, C6) e (R8, C7)). Scendiamo poi in R5… Si noti che qualora si giungesse, ad esempio, in (R5, C3), che ha valore 4, conviene salire a sinistra o in alto (aggiungendo 6, mentre osservando soltanto i valori locali si salirebbe a destra, dove c'è 1) o, cercando il massimo, a destra (aggiungendo 20, quando invece, in base ai meri valori locali, si sarebbe portati a salire a sinistra, dove c'è 5). Qualora si presentassero alternative equivalenti, come appunto in (R5, C3), sarà sufficiente ricordarne una sola. E così procedendo, una volta conclusi i calcoli sulla prima riga (R1), saremo in grado di sapere quale sarà la somma minima (o massima) che potremo totalizzare, e sapremo anche da quale casella di R1 iniziare il percorso e in quale direzione risalire ad ogni passo!
Completate voi lo schema! Si trovano due diversi percorsi a somma minima 26. [In totale, si fanno soltanto $(r - 1)\cdot(3c - 2)$ operazioni locali, nel nostro caso 154.]

**78.** Con "problema decisionale" s'intende un problema la cui soluzione è binaria: o "sì" o "no".

**80.** Con $v_{max}$ si intende, ovviamente, il massimo dei valori degli $n$ oggetti.
Ma che cosa significa *NP-hard* (NP-arduo, o NP-difficile, in italiano)? NP sta per "non-deterministico polinomiale", e vuol dire che questo problema potrebbe essere risolto in tempo polinomiale da un ipotetico algoritmo tale che, se una soluzione del problema esiste, ad ogni passo compie la scelta giusta per arrivare alla soluzione più vicina, mentre, se il problema non ammette soluzione, ad ogni passo fa la scelta che conduce al fallimento più lontano… All'atto pratico, ai fini del nostro discorso: è proprio uno di quei problemi per risolvere i quali non disponiamo di un algoritmo tempo-polinomiale nella lunghezza in bit dei dati di input, e non sappiamo neppure se un tale algoritmo esista.
Tuttavia, in questo caso specifico, ciò è vero in un senso un po' meno forte rispetto ad altri, come ad esempio i problemi dell'imballaggio e del commesso viaggiatore, per i quali nemmeno disponiamo di algoritmi pseudo-polinomiali.

**81.** Un altro problema in cui si rivela preziosa la programmazione dinamica è la ricerca della più lunga sottosequenza che due sequenze (in questo caso, due stringhe di caratteri) hanno in comune; non si richiede che i caratteri della sottosequenza siano consecutivi in qualcuna delle due sequenze date, ma che sia rispettato lo stesso ordine in entrambe…

**84.** Come si vede, lo schema risultante è analogo a quello di *Knapsack*.

**85.** Uno tra i più famosi problemi (e pure, in generale, tra i "più impegnativi", almeno in linea di principio) che *si sospettano* intrattabili è quello (di ottimizzazione) del commesso viaggiatore.
Dato un grafo pesato arbitrario, si deve trovare, se c'è, un percorso chiuso (un ciclo, o per dir meglio circuito se il grafo è orientato) che tocchi una e una sola volta ciascun nodo (ossia, *hamiltoniano*) e che sia più breve possibile (ottimo).
Rimane ugualmente impegnativo anche soltanto stabilire *se esiste* un cammino (anche non chiuso) hamiltoniano.

**89.** Nel 2016, per la gara Bebras dell'Informatica, il gruppo della Svizzera propose lo scenario qui raffigurato per un quesito, il cui spunto era dato dalla solita comunità di castori che deve decidere in quali dei dieci punti allestire tre presidî di pronto soccorso, in modo tale che da ciascuno degli altri punti sia possibile raggiungere un pronto soccorso nuotando per un solo tratto di canale.
In generale, quando si deve trovare in un grafo non orientato un insieme di nodi di cardinalità minima, tale che ciascuno degli altri nodi sia collegato da un arco con almeno uno di questi, si dice che si deve risolvere un problema di *minimo insieme dominante* (*Minimum Dominating Set*), che è NP-arduo.

Ma, sullo stesso scenario, poniamoci un'altra domanda: poiché da ciascun punto si possono controllare tutti i tratti di canale che hanno un'estremità in quel punto, qual è il numero minimo di punti nei quali installare una torretta d'avvistamento, affinché tutti i tratti di canale siano sorvegliati? In generale, questo è detto problema di *minima copertura per nodi* (*Minimum Vertex Cover*), pur'esso NP-arduo…

**95.** Con $K_{n,n}$ si denota il grafo bipartito con *n* nodi in ciascuna delle due parti, e ogni nodo di una parte collegato con un arco a ciascun nodo dell'altra parte. Nel 1930, il matematico polacco Kazimierz Kuratowski dimostrò che *ogni* grafo non planare contiene, come *sottografo*, $K_{3,3}$ o un grafo completo di 5 nodi, $K_5$; questi ultimi sono non planari: se provate a disegnarli, pur prestando attenzione, non potrete fare a meno di incrociare l'ultimo arco con uno di quelli già tracciati!

**98.** Il problema di *minima copertura di un insieme* (*Minimum Set Cover*) consiste nella scelta del minor numero di insiemi (da una data famiglia di insiemi) la cui unione coincida con un dato insieme "universo" (che spesso non è altro che l'unione dell'intera famiglia).

**99.** In che senso possiamo affermare che *Minimum Set Cover* è *equivalente* a *Minimum Dominating Set*? Se consideriamo l'impegno di tempo richiesto per risolverli esattamente, possiamo affermare che sono parimenti difficoltosi; più precisamente, si può trasformare un'istanza dell'uno in un'istanza dell'altro in tempo polinomiale. Mostriamo come il secondo si riduce al primo: semplicemente, ad ogni nodo associamo l'insieme di nodi costituito dal nodo stesso e dai suoi vicini, raggiungibili percorrendo un arco; una volta ricoperto col minor numero di tali insiemi l'insieme di tutti i nodi del grafo, basterà considerare − come soluzione di *Minimum Dominating Set* − i nodi a cui sono associati gli insiemi usati per la copertura. Lascio a voi il compito − un poco più impegnativo! − di ricondurre, in generale, *Minimum Set Cover* a *Minimum Dominating Set*.
Oltre che a dimostrare l'equivalenza (quanto a onere computazionale) dei problemi elencati in una "lista circolare" (riconducendo, in modo efficiente, ciascuno di essi al suo successivo), in informatica spesso può tornar comodo trasformare un problema in un altro ad esso equivalente (o più generale), per risolvere il quale già si dispone di un software apposito; resta inteso che poi bisogna sapere anche come interpretare (facilmente) i risultati per ottenere la soluzione del problema originario!

**100.** Se consideriamo il complemento di una minima copertura per nodi, rispetto all'insieme di tutti i nodi del grafo, otteniamo una soluzione di un altro interessante problema di ottimizzazione: trovare il *massimo insieme indipendente* (*Maximum Independent Vertex Set*), un insieme di nodi di cardinalità massima che, a due a due, *non* sono collegati da un arco…

**101.** Il grafo complementare di un grafo *G* ha gli stessi nodi di *G*, nessuno degli archi di *G*, e tutti gli archi (tra due nodi diversi) che *non* sono in *G*. Un insieme indipendente è costituito dai nodi di un sottografo completo (*clique*) nel grafo complementare, e pertanto ciascuna soluzione di un problema di massimo insieme indipendente è anche soluzione del problema di *Maximum Clique* sul grafo complementare, e viceversa: si tratta dunque di problemi equivalenti, quanto a onere computazionale.

**102.** In effetti, per il problema del massimo insieme indipendente, i migliori algoritmi finora ideati hanno una complessità rispetto al tempo di ordine $c^n$, dove *n* è il numero di nodi del grafo e *c* una costante di poco maggiore di 1.2.

**103.** Mi sembra istruttivo, oltre che divertente, accennare ai *puzzle* (o rompicapi) la cui soluzione può essere trovata, più o meno laboriosamente, con l'ausilio di un computer.
Certuni sono facilmente "programmabili" con la tecnica del *backtracking*, basata su un'idea invero un po' brutale: si procede per tentativi, alla peggio si arriva a un punto morto, allora si torna indietro fino all'ultimo tentativo fatto e se ne fa un altro, e se non ci sono altre possibilità, allora si torna ancora indietro di un passo, e così via; prima o poi, o si arriva a una soluzione o si sono tentate inutilmente tutte le strade possibili, e allora vuol dire che la particolare istanza del problema non ammette alcuna soluzione. Alla fine, se c'è almeno una soluzione, quella trovata è interamente contenuta nello "stato

finale" (magari avendo introdotto qualche piccolo accorgimento, come nel caso dell'uscita da un labirinto), e non importa sapere *come* ci si è arrivati. 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 di questo tipo di puzzle sono il Sudoku, il giro del cavallo su una scacchiera, i classici puzzle con tessere a incastro (senza conoscere il disegno finale o la disposizione finale delle tessere): lo stato finale, non noto, al quale si deve giungere, è proprio la soluzione del problema!
(Forzando la continuazione del procedimento anche dopo aver trovato una soluzione, si può fare una ricerca esauriente di *tutte* le soluzioni, purché il tempo richiesto non divenga proibitivo.)

**105.** Il programma che ho scritto non applica alcuna regola di deduzione, bensì procede esclusivamente per tentativi, e non cerca neppure una casella che presenti il minor numero di possibilità, s'intende almeno due (accorgimento che comunque, in generale, non minimizza il numero complessivo di tentativi), bensì prende la prima casella che trova con almeno due possibilità: ogni volta che assegna un numero a una casella, propaga ricorsivamente gli effetti di tale assegnazione, mediante l'eliminazione di quei numeri che rimangono fissati, dalle altre caselle dei rispettivi riquadri/righe/colonne.
Qualora rimanga almeno una casella senza alcuna possibilità di essere riempita con un numero (ed è sufficiente questo controllo per decidere il fallimento), il programma ripristina lo stato che precedeva l'ultimo tentativo fatto e procede con un altro tentativo (sulla stessa casella). Si osservi l'applicazione "ricorsiva" (in giallo) all'interno del ciclo (in azzurro) che considera a uno a uno tali tentativi.
Se lo schema può essere risolto, allora prima o poi una soluzione è trovata: basta controllare che sia rimasta una sola possibilità per ciascuna casella!

**107.** Se, ad esempio, aggiungessimo un nono sottoinsieme, costituito da 1, 2 e 3, allora le soluzioni sarebbero due; tre, se aggiungessimo anche il sottoinsieme formato da 1 e 2. Naturalmente, non è detto che un'istanza di questo problema ammetta necessariamente soluzione; potrebbe non averne!
I dati per un programma al computer possono essere semplicemente messi nella forma di una matrice di bit (cifre binarie).

**113.** Il merito di aver introdotto i "pentamini", resi celebri da Solomon W. Golomb (e da Martin Gardner) negli anni '50, va tuttavia riconosciuto all'inglese Henry Ernest Dudeney, uno dei maggiori inventori di enigmi e giochi matematici di tutti i tempi. Questa illustrazione comparve nella sua prima raccolta, pubblicata nel 1907.
Dudeney, così come lo statunitense Sam Loyd, aveva l'abitudine di aggiungere una storiella alla spiegazione del gioco. In questo caso, l'autore riporta un aneddoto che si riferisce a Roberto ed Enrico, due dei figli di Guglielmo I il Conquistatore, re d'Inghilterra. Siamo nella seconda metà del secolo XI; i due fratelli, in visita a Costanza presso la corte francese, trovano pure il tempo per svagarsi. Enrico gioca a scacchi con Luigi, Delfino di Francia, e lo vince ripetutamente, finché Luigi, innervosito oltre ogni limite, lancia i pezzi sul viso di Enrico, il quale a sua volta, benché trattenuto dal fratello Roberto, riesce a rompere la scacchiera sulla testa di Luigi…
In seguito, Enrico non solo tolse il ducato di Normandia al fratello primogenito Roberto, ma pure, approfittando dell'assenza di questi impegnato nelle Crociate, gli usurpò il trono alla morte del fratello terzogenito Guglielmo II il Rosso, succeduto al padre, e divenne Enrico I, re d'Inghilterra… ma questa è (un'altra) storia!
Dudeney immagina che la scacchiera si sia rotta nei tredici frammenti, tutti di forma diversa (i 12 pentamini più il tetramino quadrato), che sono mostrati nell'illustrazione; invita il lettore a costruirli, ritagliandoli da un cartoncino quadrettato, e gli assicura un divertente passatempo nel cercare di ricomporre la scacchiera originaria − senza tuttavia precisare se le caselle siano bianche o nere anche sul retro, e se quindi i vari pezzi possano essere ribaltati o meno… Dudeney ne dà una possibile soluzione, senza ribaltamenti. A voi il piacere di trovarne almeno una!

**114.** … Infatti, le figure in basso non si possono ottenere per rotazione dalle corrispondenti in alto!
I *pentamini* sono le 12 figure che si formano con 5 quadratini di ugual lato, ove ciascun quadratino ha un lato in comune con almeno un altro; 6 di queste figure devono essere colorate da entrambe le parti, per poterle anche ribaltare, mentre per far assumere tutti i possibili orientamenti alle altre 6 è sufficiente poterle ruotare.

**115.** Ecco un classico puzzle con i pentamini, oggetto di uno dei primi programmi che hanno impiegato la tecnica del *backtracking*, progettato dall'eminente logico Dana S. Scott, il quale lo realizzò nel 1958 con l'aiuto di Hale F. Trotter, proprio allo scopo di calcolare tutte le soluzioni essenzialmente differenti di un puzzle combinatorio con i 12 pentamini: si trattava di formare un quadrato 8×8, con un "buco" 2×2 al centro, come mostrato nella figura…
Perché fu deciso di non ribaltare proprio il pentamino P (arancione), quando è considerata la terza possibile collocazione del pentamino X (fucsia)? Perché è quello che, sistemato così X, presenta il maggior numero di possibili collocazioni, che vengono dimezzate grazie alla simmetria NW-SE!

**117.** Sia questo puzzle, sia uno schema di Sudoku possono essere ricondotti a istanze del problema dell'*esatta* copertura di un insieme (a differenza della *minima* copertura, qui si richiede di scegliere insiemi non in minor numero, bensì a due a due disgiunti), che rimane NP-arduo…

**118.** … E quindi le soluzioni del puzzle di Dudeney saranno certamente meno di 16146.
Donald Knuth, grande nome dell'informatica, ha descritto algoritmi − che all'atto pratico hanno funzionato assai bene − proprio per risolvere questo tipo di problemi: le loro prestazioni si sono rivelate sorprendenti, persino in casi di ragguardevoli dimensioni.

**119.** Il caso 6×10 fu risolto per primo, in modo esauriente, da Colin Brian Haselgrove e sua moglie Jenifer nel 1960. Il programma di John G. Fletcher, ottimizzato per questo particolare problema e pubblicato nel 1965, comprendeva 765 istruzioni nel suo loop più interno; venne eseguito da un calcolatore IBM 7094, che aveva una frequenza di clock di 0.7 MHz e ad ogni singolo ciclo di clock accedeva a due parole di memoria di 36 bit.

**120.** L'esecuzione dei miei programmi (non ottimizzati) sui personal computer della scuola – circa una decina d'anni or sono – ha richiesto invece qualche ora, addirittura 10 ore nel caso del rettangolo 3×20 (dove ho proceduto con una ricerca bruta, senza sfruttare simmetrie).

**121.** I casi possibili sono ripartiti in classi (di cui una vuota), a seconda di come possono essere disposti i quattro pentamini Y agli angoli.

**122.** Tassellare il piano (infinito) significa ricoprirlo interamente, senza vuoti né sovrapposizioni, usando soltanto le forme (poligonali) appartenenti a un certo insieme finito: nel nostro caso, una forma soltanto, costituita da un singolo pentamino. Dei pentamini F, T e U occorre considerare anche le copie ruotate di 180°, mentre non c'è bisogno di ribaltarne alcuno.
Sebbene nell'esempio qui illustrato la forma usata non sia poligonale, il dettaglio di un affresco di Lorenzo e Jacopo Salimbeni (nell'Oratorio di San Giovanni a Urbino) rende bene l'idea di tassellatura *periodica*, che si ripete lungo due direzioni non parallele.

**123.** Veniamo al classico problema dell'uscita da un labirinto. In ogni casella (percorribile, qui segnata col carattere '.') sono al massimo quattro i passi da tentare. Onde evitare di ritornare sui propri passi cadendo in circoli viziosi, bisognerà ricordare le caselle già percorse marcandole con un carattere apposito (qui una lettera 'o') lasciato come un sassolino per ricostruire il percorso attuale dalla casella di partenza; qualora si dovesse tornare indietro, basta assegnare nuovamente il carattere '.' alla casella che si lascia, però – per maggior efficienza – conviene assegnarvi un altro carattere (ad esempio un asterisco, da tener nascosto, sostituendolo con il punto soltanto al momento della stampa finale) allo scopo di ricordare che su quella casella si è già passati, ma senza successo. È ovvio che, se il problema ammette più soluzioni, quella trovata dipende dall'ordine in cui si sono esplorate sistematicamente le quattro direzioni consentite.

**127.** Ci sono poi puzzle un po' più complicati da programmare, ove una soluzione (se c'è) consiste nella sequenza di transizioni (o mosse), non nota, che ha portato dallo stato iniziale a quello finale, di successo, magari già noto a priori (vale a dire che si sa dove si vuole arrivare, ma non si sa come); inoltre, potrebbe esservi il rischio di incorrere in "cicli"… In taluni casi, si rivela interessante cercare la soluzione più breve, o una delle più brevi, o addirittura tutte le più brevi.

**128.** Vediamo, come esempio, il gioco del 16 inglese (un altro, famoso, è il gioco del 15, che qui menziono soltanto, invitando chi è interessato a indagare sulle configurazioni più lunghe da risolvere e sulla ricerca euristica di una delle soluzioni più brevi). Qui è riportata l'illustrazione originale che accompagnava la presentazione fatta da Sam Loyd ai suoi lettori. Egli − alla pari di Dudeney − aveva l'abitudine di aggiungere una storiella alla spiegazione. Di questo gioco, che chiamò *Fore and Aft* ("A prua e a poppa", come dire "Avanti e indietro"), scrisse che fu inventato da un marinaio inglese, il quale trascorse gran parte della vita sull'isola di Staten Island, nella baia di New York, dove ai turisti vendeva dei giochi da lui stesso intagliati nel legno. In particolare, costui era orgoglioso proprio di questa sua creazione, che fu portata in Inghilterra e là divenne assai popolare col nome di "gioco del sedici inglese". In realtà, questo gioco − da annoverare fra i tanti che ricordano il solitario classico, alla francese o all'inglese − pare sia stato pubblicato per la prima volta dall'inglese Walter William Rouse Ball nel 1892, e certo è che fu in auge in tarda età vittoriana, per cui in Italia divenne noto come "piccolo solitario vittoriano".

**129.** Le pedine nere possono muoversi soltanto verso nord (N) o verso ovest (W), le bianche soltanto verso sud (S) o verso est (E); non sono permessi movimenti in diagonale. Una pedina può spostarsi di una casa, oppure può scavalcare una pedina di colore opposto, per andare ad occupare la casa libera. Lo scopo è lo scambio di posti tra le pedine dei due colori.
In virtù di queste regole, durante una "partita" può venire a crearsi una situazione di blocco, ma non può accadere il ripetersi di una stessa posizione: pertanto è inutile controllare se lo stato raggiunto è già stato visitato. Poiché in ogni momento c'è una sola casa libera e poiché non sono permesse mosse in diagonale, né a ritroso, ne consegue che ogni volta che si deve fare una mossa, non più di due pedine di ciascun colore hanno la possibilità di muoversi.
Vi sono davvero tante soluzioni − ma quante saranno? − ovviamente simmetriche a gruppi di quattro, a seconda che inizi il bianco o il nero, ciascun colore con l'una o con l'altra delle pedine che possono iniziare.
W. W. Rouse Ball, riportandolo nella sua opera enciclopedica *Mathematical Recreations and Essays* del 1892, ne dà una soluzione in 51 mosse; nella terza edizione, del 1896, ne dà un'altra di 48 mosse. Ma nella quinta edizione, del 1911, pubblica quella che egli dice trovata da Dudeney nel 1898: essa consta di sole 46 mosse ed è particolarmente elegante per la sua simmetria.
Soltanto alcuni decenni fa è stato provato che 46 mosse sono necessarie e che esistono più soluzioni minime (in 46 mosse); ho scritto un programma che ha stampato tutte le 2476 soluzioni minime che iniziano con la mossa 1, impiegando circa 11 minuti (su un computer ormai vecchiotto).
Com'è descritta precisamente una soluzione? Senza creare ambiguità, possiamo denotare ciascuna mossa indicando il numero corrispondente alla casa da cui parte la pedina che si muove. Facciamo iniziare il nero con la pedina che sta nella casa 1 (sebbene non necessario, in neretto sono evidenziati i salti). Su ciascuna linea sono scritte 23 mosse: a metà partita la posizione dei due colori è simmetrica. Se prendiamo le prime 22 mosse, le ribaltiamo e cambiamo i segni, otteniamo le prime 22 della seconda linea! Anche la distribuzione dei salti sulle due linee è simmetrica.
Siccome, in questo solitario, una posizione non può ripetersi, ha pure senso chiedersi di quante mosse siano costituite le soluzioni *più lunghe*: la risposta è 58 (e sono 75 quelle che iniziano con 1).

**130.** E veniamo infine ai giochi tra due avversari, in cui un computer può essere programmato per giocare (possibilmente al meglio) contro un umano o contro un altro software.

MasterMind è un notissimo gioco, ma di altra natura. Spesso agli allievi si propone di scrivere un programma che giochi il ruolo del codificatore. Nella classica versione con sei colori e codice segreto di quattro colori con possibili ripetizioni, le possibili sequenze sono $6^4 = 1296$. Un ovvio programma che sostenga invece la parte del solutore potrebbe iniziare con un tentativo scelto a caso tra i 1296; acquisita la risposta del codificatore, elimina tutti i codici incompatibili con la risposta ricevuta e, tra quelli rimasti, ne sceglie uno a caso; e così via, finché non perviene al codice segreto. Tuttavia, questa strategia non garantisce la minimizzazione del numero massimo di tentativi!

**131.** Il NIM è il più classico dei giochi con i fiammiferi. Si tratta di un gioco *combinatorio imparziale* (nessuna distinzione di "materiale" tra i due giocatori, nessuna possibilità di pareggio), analizzato dal matematico statunitense Charles L. Bouton all'inizio del '900. Si dispongono dei fiammiferi in diverse file; il giocatore di turno deve togliere da una fila a sua scelta un numero qualsiasi di fiammiferi, al minimo uno, al massimo tutti; vince chi toglie l'ultimo fiammifero. Per "risolvere" il gioco, si scrivono in colonna i numeri in binario dei fiammiferi in ciascuna fila e poi si calcola, per ogni colonna di bit, un bit di parità sugli 1 (ciò che corrisponde a fare l'*or esclusivo*). Se i bit di parità sono tutti 0, non c'è mossa che conservi tale proprietà; in caso contrario, almeno una può portare tutti i bit di parità a 0, e pertanto chi si trova in quest'ultima situazione vince, poiché perde chi rimane senza fiammiferi (e dunque con tutti i bit di parità ovviamente a 0). Realizzare un programma che giochi a NIM (magari lasciando qualche *chance* all'avversario!) non dovrebbe presentare eccessive difficoltà…

**133.** Vi sono giochi "infiniti": se nessuno dei due avversari commette errori, la partita non termina.
Nel *Kayles* normale (messi in riga $n$ birilli, ciascun giocatore, a turno, può buttar giù un birillo o due birilli adiacenti; chi butta giù l'ultimo vince) ha la strategia vincente il primo giocatore, per ogni $n > 0$.
Quello che ho chiamato *Babylone-one* è un esempio di *falso gioco*, in cui entrambi i giocatori sono "ininfluenti": indipendentemente dalle loro scelte, il numero di mosse e l'esito finale sono determinati soltanto dal numero di gettoni posti sul tavolo. Ad ogni mossa, il giocatore di turno deve impilare due pile di stessa altezza, e se non può farlo perde.
All'opposto, il gioco ideato da Patrick M. Grundy nel 1939 non è un falso gioco: partendo da una pila di $n$ gettoni, il giocatore di turno deve spezzare una pila a sua scelta in due pile di *diversa* altezza, e se non può (perché sono rimaste soltanto pile di uno o due gettoni) perde. Per ogni $n$ si può stabilire quale dei due avversari abbia la strategia vincente; è stato congetturato che la successione che esprime per ogni $n$ quale dei due giocatori possa vincere diventi prima o poi periodica, ma ancora non si sa…

**137.** Il popolare gioco del tris, forse risalente all'Antico Egitto, è stato oggetto del primo videogame (su EDSAC, 1952). Si tratta di un gioco "alla pari": se nessuno dei due commette errori, nessuno vince! In una variante, proposta dal prolifico Martin Gardner, si aggiunge una casella alla riga orizzontale in basso, sulla destra, e per vincere su questa riga più lunga bisognerà occupare tutte e quattro le caselle, mentre altrove basta il tris, anche sull'ulteriore diagonale di tre caselle che viene a crearsi…

**138.** Ecco altri giochi della stessa classe del tris (finiti, a informazione perfetta, a due competitori che si alternano a muovere, a somma zero), ma assai più complessi. Per tutti, in teoria, si può stabilire quale si verifica dei tre casi possibili già previsti nel 1912 dal matematico tedesco Ernst Zermelo, fondatore della teoria assiomatica degli insiemi: salvo errori, è *sempre* in grado di vincere chi inizia, oppure chi risponde, oppure è patta teorica; ma soltanto per alcuni di essi ciò è possibile all'atto pratico.
In qual modo può essere "risolto" un gioco? Un esempio di quella che si dice risoluzione in senso ultra-debole fu dato nel 1948 da John F. Nash (allora studente a Princeton, futuro premio Nobel per l'economia nel 1994), con una dimostrazione per assurdo, e non costruttiva, del fatto che a Hex dev'essere il primo giocatore ad avere una strategia vincente (pur ignorando quale essa sia).
La dama sulla scacchiera tradizionale 8×8 è stata risolta (in senso debole) nel 2007 (vi ha lavorato una cinquantina di computer per 18 anni!) e si tratta del più grande gioco per il quale finora sia stato trovato un risultato di questo tipo: se nessuno fa errori, la partita finirà patta. Per fortuna, sapere che la dama è "alla pari" non toglierà comunque il piacere di giocare una partita!

Che cosa significa “in senso debole”? È stato determinato il risultato (di parità) e individuata una strategia corretta (non perdente) a partire dalla posizione *iniziale*: per far questo è “bastata” l’analisi di circa quarantamila miliardi di posizioni: una parte su 12.5 milioni, rispetto al totale delle possibili posizioni sulla scacchiera!

**139.** Oltre una decina d’anni fa, a scuola, grazie all’impegno di un mio eccellente allievo, abbiamo realizzato un programma che gioca ad Awari, un antichissimo gioco africano, tuttora diffuso in gran parte del mondo, con tantissime varianti. Le regole sono semplici, il gioco affascinante, con forti valenze rituali e simboliche − si può dire religiose − oltre che didattiche.
Noi abbiamo seguito la variante studiata, a partire dagli anni ’90, da un gruppo di ricercatori olandesi, che nel 2002 hanno risolto “in senso forte” tale gioco: per *ognuno* degli stati possibili (qui quelli essenzialmente differenti sono circa 900 miliardi) hanno trovato la lista delle mosse “giuste” da fare; tutto quanto è stato poi memorizzato in un database che occupa 778 GB, dopodiché – giunti a questo punto – la stesura di un programma imbattibile è stata assai semplice!
Come si gioca? Il tavoliere ha, per ciascuna delle due parti, una fila di 6 buche (contenenti all’inizio 4 semi ciascuna) e un “granaio” (che conterrà i semi catturati). Il giocatore che deve muovere sceglie una delle proprie buche, non vuota; preleva tutti i semi che vi sono contenuti e, procedendo in senso antiorario, li semina a uno a uno nelle buche successive, anche nel campo opposto, saltando poi la buca originaria se i semi da questa prelevati sono più di 11: fatta questa operazione, tale buca rimane quindi sicuramente vuota. Se l’ultimo seme cade in una buca avversaria che contiene o 2 o 3 semi (compreso quello appena seminato) questi sono catturati, e così di seguito, a ritroso, finché si incontrano buche nel campo avversario con 2 o 3 semi.
Ovviamente, non è detto che la mossa che porta al massimo risultato immediato sia la migliore: spesso una tattica *greedy* conduce alla sconfitta, se l’avversario gioca bene!
Se tutte le buche nel campo del giocatore di turno sono vuote, egli non può muovere; allora l’avversario trasferisce nel proprio granaio i semi che sono eventualmente rimasti nel proprio campo, e la partita finisce. Il punteggio è determinato dalla differenza tra i contenuti dei rispettivi granai.
Questa sarebbe la regola più semplice per stabilire la fine di una partita; tuttavia, per prolungare il gioco, nell’Awari qui considerato vige una regola *fair* (leale): non sono consentite mosse che lasciano l’avversario senza semi, eccetto nel caso in cui *tutte* portino inevitabilmente a questa situazione. (Non è permesso comunque saltare un turno, ciò che invece è previsto in altre varianti.)
Infine, se una stessa posizione si ripete per tre volte, la partita termina: secondo la nostra variante, i semi rimasti sono divisi *esattamente* a metà tra i due giocatori, e aggiunti ai rispettivi granai.
Secondo l’approccio più tradizionale, noi abbiamo seguito l’idea di analizzare tutti gli stati raggiungibili, a partire dalla posizione attuale, in un certo numero prefissato di mosse: quindi su un orizzonte necessariamente limitato. Sarebbe opportuno procedere oltre la profondità massima stabilita qualora lo stato raggiunto non sia *quiescente*: ad esempio, nell’Awari, uno stato può dirsi quiescente se non vi sono buche con semi esposti a cattura alla mossa successiva.

**140.** L’idea di base è questa: ciascuno dei due avversari cercherà di minimizzare la massima perdita per lui possibile, secondo un criterio “condiviso”!
Il punto delicato è la valutazione degli stati (non finali) nei quali l’analisi si ferma e non procede oltre…
Nel caso dell’Awari, abbiamo considerato semplicemente la differenza tra i contenuti dei due granai: quello del giocatore *al quale spetterebbe muovere nello stato raggiunto* meno quello dell’avversario. In generale, la progettazione della funzione di valutazione di una posizione (pur quiescente) è piuttosto delicata, poiché si assume implicitamente che ciascuno dei due contendenti giochi al meglio delle proprie possibilità proprio in base a *questa* particolare funzione di valutazione.

**141.** Applicando poi le classiche tecniche di “potatura”, che evitano l’esplorazione di quelle alternative che *sicuramente* non sono migliori di una già analizzata, il tempo richiesto dall’analisi si può ridurre sensibilmente (nel nostro Awari, a profondità 8, è sceso sotto la metà).

Il primo procedimento di potatura, conosciuto come α-β, fu presentato da John McCarthy nell'estate del 1956, durante il primo seminario sull'*intelligenza artificiale*; tra le altre cose, proprio in quella circostanza McCarthy suggerì le idee in base alle quali poi progettò il linguaggio LISP.
Un'approssimazione *branch-and-bound* dell'algoritmo di potatura α-β fu codificata e utilizzata per la prima volta in un programma per gli scacchi, finito nel 1958, da Allen Newell, Herbert A. Simon (futuro premio Nobel per l'economia nel 1979) e J. C. (Cliff) Shaw, all'epoca ricercatori presso il Carnegie Institute of Technology di Pittsburgh, Pennsylvania.

**142.** Queste tecniche di potatura, insieme ad alcune più moderne (che sperimentammo con l'Awari, ottenendo ancora maggior efficienza), sono dettagliate (a livello di codice in C++) nel mio libro "Dai giochi agli algoritmi", pubblicato nelle Edizioni Kangourou Italia. La seconda edizione (con copertina viola), riveduta e ampliata nel 2019, potete scaricarla gratuitamente in formato pdf (di 482 pagine):

https://drive.google.com/file/d/1tMllvuRregMCwwYb-c25DlhJ7FxsPaM8/view?usp=sharing

Nel volume troverete approfondimenti sui temi e problemi menzionati in questa sede e, mi auguro, tanti altri spunti per stimolare e rendere ancor più piacevole l'apprendimento di qualche principio dell'informatica ai vostri allievi.

**143.** Più di un secolo è trascorso dalla costruzione di quello che probabilmente è stato il primo automa logico interattivo realizzato con successo, sebbene la procedura seguita avesse parecchi limiti: infatti, *non* garantisce la vittoria del Bianco in tutte le possibili posizioni di partenza (legali, col tratto al Bianco); in certi casi, il Bianco si trova bloccato, poiché l'azione prevista non può essere compiuta oppure è illegale, o perde la Torre o giunge sì a dare il matto, ma in un numero di mosse maggiore di 50, ciò che, secondo le regole, conduce alla patta: in effetti, questa semplice procedura – peraltro ingegnosamente progettata e realizzata – non è generale e nemmeno ottima, e l'inventore ne era affatto consapevole.
Non molti anni fa sono stati realizzati approcci radicalmente diversi alla progettazione di giocatori artificiali, basati su reti neurali "profonde". Un aspetto chiave – e un po' inquietante – di questa tecnologia è che i criteri con cui è condotta una partita non sono direttamente comprensibili, nemmeno da parte di chi ha progettato la rete neurale: non si può sapere *perché* la rete abbia calibrato i suoi pesi o parametri, che si contano a milioni, proprio su quei certi valori; non sappiamo *come* abbia "ragionato" per giungere a quei risultati, e quindi – in sostanza – l'algoritmo non può essere descritto, né tantomeno spiegato…

**144.** Chiudo in bellezza, spiegando un algoritmo di elaborazione di immagini – piuttosto semplice da programmare – che ha il fascino di un gioco di prestigio… Col termine *photomaton* furono chiamate quelle popolari macchine fotografiche automatiche, che stampano quattro esemplari della stessa fotografia in formato tessera; apparvero in Francia, per la prima volta, nel 1925. Se i pixel di un'immagine bidimensionale si ridistribuiscono ad ogni passo come illustrato in figura, si ottengono quattro figure "simili", ma non identiche poiché i pixel sono soltanto spostati, non modificati… Se le dimensioni sono entrambe pari, dopo un numero di passi che dipende funzionalmente da queste dimensioni, riapparirà l'immagine originale – e, nel frattempo, certi passi potrebbero aver prodotto effetti sorprendenti!

Se desiderate ulteriori spiegazioni, materiale o suggerimenti, scrivetemi!

Genova, 11 settembre 2023

Lorenzo Repetto

repetto@calvino.ge.it

lorenzo.repetto@calvino.edu.it