SOLUZIONE DEI PROBLEMI E PIANIFICAZIONE

Maurizio Matteuzzi

It has come to my attention that every time we solve a problem,
we create two more.
From now on, all problem solving is forbidden

La risoluzione dei problemi è in un certo senso l'anima, la quintessenza dell'IA; il modello fisico simbolico di Simon e Newell è stato ciò che ha caratterizzato i primordi della disciplina, e il suo contraltare algoritmico è stato il GPS, il progetto di un programma in grado di risolvere, in senso del tutto generale, i problemi, ovvero, qualsiasi problema. Questo approccio, evidentemente troppo ambizioso, fu poi nelle fasi successive abbandonato, e sostituito da quello dei così detti "expert systems", sistemi riferiti a un "micromondo", molto meno ambiziosi dunque, ma più prossimi ad una realizzazione concreta. Ciò non di meno, il GPS costituisce di fatto una pietra miliare dello sviluppo della nostra materia, e un termine di paragone ineludibile.

La prima questione da affrontare è come si formalizza un problema. Se un problema non è opportunamente formalizzato, è chiaro che non potrà mai essere "trattato" con mezzi informatici. Il tema della rappresentazione della conoscenza rimane centrale in tutte le applicazioni di intelligenza artificiale[1].

Le esigenze del problem solving, da questo punto di vista, si incentrano sulla necessità di rappresentare il problema che si vuole risolvere. Una base matematica estremamente utile per questo aspetto, e in un certo senso imprescindibile in quanto ne costituisce la strumentazione essenziale, è la teoria dei grafi. Si faccia bene attenzione a non confondere "grafo" con "grafico" (che disgraziatamente si dicono in inglese tutti e due "graph", il che spesso ingenera confusione nell'abbondantissima letteratura tradotta): un grafico è una

[1] Per una discussione e una trattazione approfondita si rimanda a Pezzulo (in questo volume).

rappresentazione, schematica o pittorica, ma sempre di tipo "iconico", ovvero la cui espressività è basata sulla verosimiglianza, ossia la similitudine del segno con l'oggetto denotato. Un grafo, viceversa, è una precisa struttura matematica. Sia dato un insieme di "vertici", che chiameremo V; sia dato un insieme di "spigoli", che chiameremo S. Allora un grafo è una funzione iniettiva di S sul prodotto cartesiano di V per se stesso. Intuitivamente ciò significa che, data una coppia di vertici qualsiasi, non è detto che ci sia lo spigolo che li collega. È facile, da questa definizione, passare a una forma di rappresentazione grafica. Tracceremo uno spigolo di collegamento tra quei vertici che sono correlati:

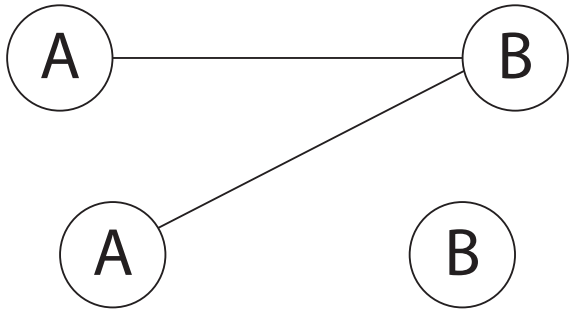

Figura 1

Un grafo è quindi il modo canonico per rappresentare una relazione: gli elementi diventano i vertici, e si tracciano gli spigoli colleganti quelle coppie di vertici per i quali la relazione vale. Così, se abbiamo Aldo, Brando, Carlo, Damiano ed Ermanno, e Carlo è figlio di Aldo, mentre Damiano è figlio di Ermanno, il grafo corrispondente prenderà la seguente forma:

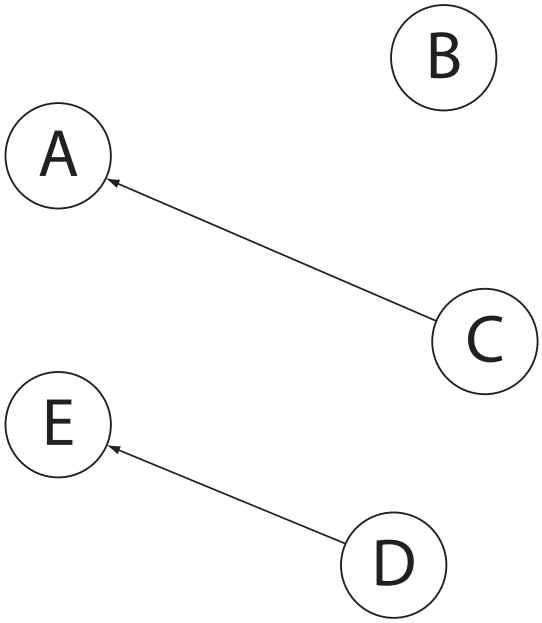

Figura 2

È ovvio che in questo modo noi rappresentiamo una relazione in senso matematico, cioè un insieme di coppie, eventualmente ordinate; questo significa che la distanza e la forma dei collegamenti perdono di significato dal nostro punto di vista. E significa anche, quindi, che si possono dare forme apparentemente diverse che in realtà costituiscono lo stesso grafo:

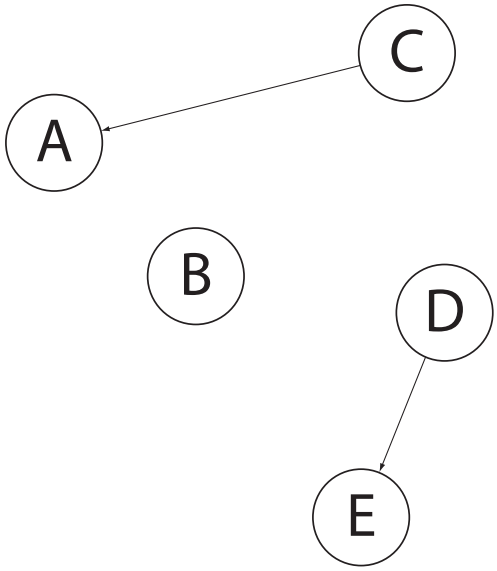

Figura 3

Ciò vale a dire che un grafo può essere disegnato in diversi modi, pur conservando le stesse proprietà.

La teoria dei grafi si fa risalire a un lavoro di Eulero, che vale la pena di richiamare, e che si riferisce ai "Sette ponti di Koenigsberg". La città di Koenigsberg (oggi Calininggrad), ben nota per avere dato i natali a Kant, era costituita di quattro parti, due isolette sul fiume e le due sponde dello stesso. Tra queste quattro parti si collocavano sette ponti, secondo la figura seguente

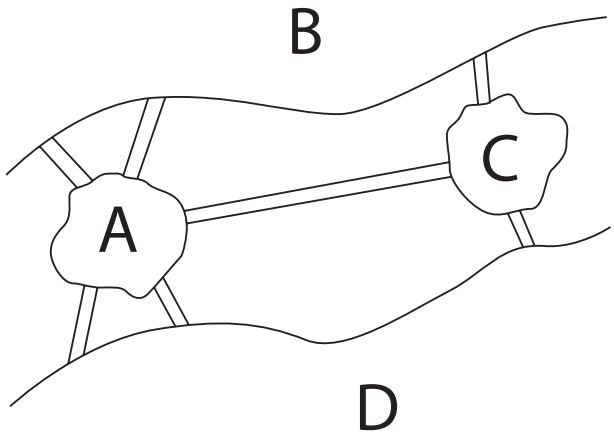

Figura 4 – I ponti di Koenigsberg in forma stilizzata.

Ora, venne posto ad Eulero il problema che non si riusciva a visitare tutta la città passando una e una sola volta per ciascun ponte e ritornando al punto iniziale. Il ragionamento di Eulero è il seguente: se devo visitare una zona, ogni volta che c'è una entrata ci deve essere anche un'uscita. Dunque il numero dei collegamenti deve essere pari. Ragioniamo in termini di grafi:

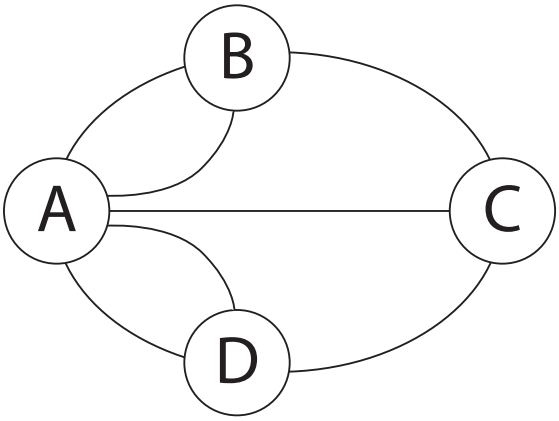

Figura 5 – Grafo dei ponti di Koenigsberg.

In termini moderni oggi diremmo: "In un grafo, esiste una linea di Eulero (cioè un percorso che collega tutti i nodi attraversando ogni spigolo una sola volta) se e soltanto se i nodi sono tutti di grado locale pari". Come si vede, nel grafo che rappresenta il problema di Koenigsberg, tutti e quattro i nodi hanno grado locale dispari, rispettivamente 5, 3, 3, 3. Analogamente, un percorso che colleghi tutti i nodi partendo da un nodo A e finendo in un nodo B, ciò che si chiama "linea di Hamilgton", esiste se e solo se A e B sono gli unici due nodi di grado dispari. A queste considerazioni, relativamente semplici, si riconducono tutti quei problemi che richiedono di tracciare una figura senza mai staccare la penna:

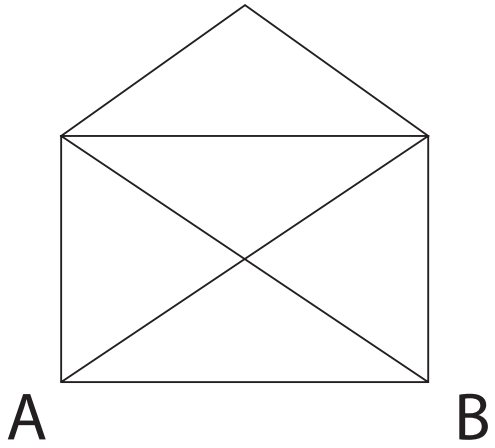

Figura 6 – Busta aperta.

Qui si vede che il problema è risolubile, a patto di partire da A o da B. Viceversa, non è risolubile l'analogo della figura che segue:

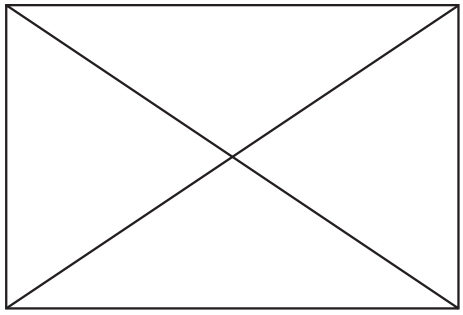

Figura 7 – Busta chiusa.

Per i nostri scopi, non è in questa sede necessario sviluppare la teoria dei grafi. È però importante che se ne capisca la natura, e che ci si familiarizzi con un particolare tipo di grafo, che gioca un ruolo essenziale nel problem solving: l'albero. Un albero si definisce come un grafo aciclico, ossia privo di cammini che tornino su se stessi, ciò che in gergo informatico si chiama "loop". I grafi alberi presentano diverse proprietà interessanti dal nostro punto di vista. L'assenza di loop può infatti essere sfruttata per l'importante caratteristica che, dato un nodo, esiste ed è unico il percorso che lo collega al nodo radice. Ogni nodo ha infatti un solo nodo padre, mentre può avere un numero qualsiasi di figli. Così questa proprietà, della unicità di percorso, è utilizzata dai sistemi operativi per rendere univoca la denominazione di un file. La metafora delle cartelle ("directories") che contengono altre cartelle, o files, oggi universalmente adottata dal software di base, costituisce in realtà l'implementazione di una struttura ad albero.

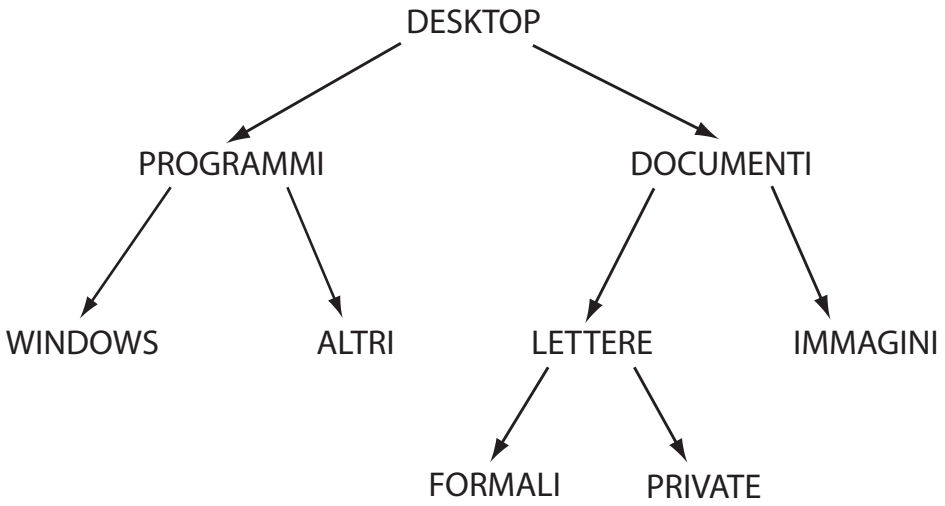

Figura 8 – Struttura ad albero di file system generico: desktop, programmi, lettere, etc.

Diverse sono le proprietà notevoli dei grafi alberi. Tra esse, va notato che un albero è una struttura ricorsiva. Se separiamo un qualsiasi nodo, con tutto quello che da esso dipende, abbiamo un sottoalbero, mentre anche la struttura decurtata è un albero. Se in un albero sostituiamo un nodo terminale, cioè una foglia, con un albero, quanto viene prodotto è ancora un albero. E questo è naturalmente molto comodo per il così detto "mounting", ovvero l'aggiunta a un file system di una nuova area di memoria. Un albero è anche una struttura strettamente gerarchica: abbiamo figli di primo livello, poi si passa ai figli dei figli, e così via. Ad ogni livello il numero dei nodi cresce, secondo un parametro che viene detto "fattore di ramificazione". Ad esempio, se pensiamo all'albero genealogico di una persona, dato che ognuno ha un padre e una madre, avremo un fattore di ramificazione 2, ossia un albero binario.

Una caratteristica notevole del grafo albero è che si presta a rappresentare in modo naturale l'andamento di una risoluzione di un problema. In un problema noi abbiamo una situazione di partenza, che viene identificata con il nodo radice, e delle possibili azioni, ossia operatori che fanno transire da uno stato ad un altro. Si parla dunque del così detto "spazio degli stati" di un problema, che si sviluppa secondo un grafo albero. Si pensi ad una partita a scacchi. Abbiamo una situazione iniziale, che è la disposizione dei pezzi sulla scacchiera, e che costituisce il nodo radice. Il bianco ha a disposizione venti possibili mosse, quindi il nodo radice ha venti figli. Il nero può rispondere a sua volta in venti modi, dunque ciascun figlio del nodo radice ha a sua volta venti figli. E così via. L'insieme teorico di tutte le possibili mosse, e quindi di tutte le possibili "partite", prende il nome di "spazio degli stati" del problema. I nodi così detti "foglia", cioè terminali, sono i possibili esiti (vince il bianco – vince il nero – stallo). Cosa vuol dire allora "risolvere" un problema? Vuol dire determinare un percorso che, partendo dal nodo radice, e applicando solo operatori ammessi, giunga ad una foglia del valore voluto (ad esempio, se gioco col bianco, ad una delle foglie di valore "vince il bianco"). Supponiamo ora che sia data una certa situazione sulla scacchiera. Ad esempio, uno dei classici problemi di scacchi, del tipo "il bianco muove e matta in due mosse". Se pensiamo all'albero dei possibili stati del caso dato, riscontreremo facilmente che esso è un sottoalbero dello spazio degli stati come definito prima. Esso prende il nome di "spazio problemico", ed è costituito da tutti i possibili nodi effettivamente raggiungibili, con gli operatori dati, a partire dal problema considerato. Infine, supponiamo adesso di cercare una soluzione per il nostro problema. È chiaro che solo nel caso più sfortunato dovremo percorrere tutto lo spazio problemico prima di trovare una soluzione. Abbiamo dunque un ulteriore sottoalbero, che prende il nome di "spazio di ricerca". Pertanto, se denominiamo:

- SS lo spazio degli stati
- SP lo spazio del problema
- SR lo spazio di ricerca,

avremo che, in generale,

$$SR \subseteq SP \subseteq SS$$

Alla luce delle considerazioni su esposte, possiamo concludere che trovare una soluzione a un problema diviene, prima di tutto, una questione di "navigazione" entro un grafo albero. Il procedimento risolutivo sarà tanto più efficiente quanto più lo spazio di ricerca sarà ridotto rispetto allo spazio problemico. In quest'ordine di idee, ecco che possiamo assumere una prima, fondamentale distinzione. Noi possiamo pensare ad un modo meccanico di navigare un grafo, seguendo regole deterministicamente predefinite, oppure assumere una strategia, che faccia appello a un qualche criterio di scelta tra le possibili vie da seguire. Nel primo caso avremo la certezza del successo, ma dovremo confrontarci con il fattore di ramificazione, ossia con la crescita esponenziale degli stati, nel secondo avremo una efficienza molto maggiore, ma ci assumeremo il rischio dell'insuccesso. La prima via prende il nome di "ricerca cieca", la seconda di "ricerca euristica".

Tecniche di ricerca cieca

Come abbiamo detto, la ricerca cieca costituisce un modo di analizzare sistematicamente, secondo un procedimento deterministico, tutti i nodi di un albero, alla ricerca della soluzione del problema. Un algoritmo di ricerca è una procedura che prende in input un problema, e restituisce il fallimento, oppure una soluzione, ossia un cammino che collega il nodo radice con una foglia voluta, o "goal".

Data la struttura di un albero, l'esplorazione può essere organizzata in profondità, cioè passando da un nodo padre a un nodo figlio, e poi al figlio del figlio e così via, scendendo sempre più verso il basso (la rappresentazione tradizionale di un albero è con la radice in alto e i nodi foglie in basso), oppure analizzando tutti i nodi dello stesso livello, per poi passare al livello successivo. Si hanno pertanto due famiglie di algoritmi. Nel primo caso parliamo di "ricerca in profondità", nel secondo di "ricerca in ampiezza". Come vedremo, ciascuna presenta comparativamente vantaggi e svantaggi.

Algoritmo di ricerca in profondità

0 Vuota la lista
1 Inserisci il nodo radice nella lista
2 SE non ci sono più nodi della lista
    ALLORA fermati: la procedura ha fallito
3 SE il nodo all'inizio della lista è un nodo finale
    ALLORA fermati: hai risolto il problema
4 SE il nodo all'inizio della lista non ha figli
    ALLORA toglilo dalla lista
    ALTRIMENTI sostituisci tale nodo con i suoi figli inserendoli all'inizio della lista stessa
5 Ritorna al passo 2[2]

[2] Cfr Fum (1994).

È opportuno notare il comportamento della ricerca in profondità quando viene raggiunto un nodo foglia che non costituisce una soluzione. In questo caso, l'algoritmo compie la così detta operazione di "backtracking", che consiste nel risalire al nodo padre, e proseguire esplorando un diverso figlio. In questa forma, il backtracking si dice "cronologico", perché si ritorna all'ultimo nodo visitato in precedenza. La scansione dell'albero avviene pertanto da sinistra a destra, vale a dire, verrà trovata la soluzione più a sinistra di tutte.

In questa versione, l'algoritmo è tuttavia esposto ad alcuni non trascurabili rischi. Il primo e più importante è che non tiene memoria dei nodi già visitati, e questo può portarlo a un loop infinto. Un primo miglioramento indispensabile è allora la così detta "detezione di cicli". L'algoritmo è sostanzialmente lo stesso, ma anziché gestire una sola lista, ne gestisce due; in una vengono elencati i nodi ancora da analizzare, nell'altra quelli già visitati. Se un nodo è già stato visitato, viene escluso dal novero delle possibilità di prosecuzione. Questa modifica non garantisce però ancora che la ricerca in profondità non prenda un andamento che va all'infinto, pur visitando nodi sempre diversi. Supponiamo ad esempio che il nostro problema sia quello di generare una terna pitagorica, cioè tre numeri tali che il quadrato del primo sia uguale alla somma dei quadrati degli altri due. Supponiamo che il nostro punto di partenza sia la terna <1, 1, 1>, e che gli operatori consentiti siano "aggiungi 1 al primo elemento", "aggiungi 1 al secondo elemento", "aggiungi 1 al terzo elemento". Il programma, applicandoli in questo ordine, genererà successivamente le terne

<2, 1, 1>
<3, 1, 1>
<4, 1, 1>
…

Si vede pertanto che l'algoritmo ha imboccato un ramo infinito dell'albero. Situazioni come questa possono essere affrontate con un algoritmo che scandaglia l'albero a livelli crescenti di profondità, ovvero per *approfondimento iterativo*.

Algoritmo di ricerca per approfondimento iterativo

0 Vuota la lista
1 Poni il limite corrente P uguale a 1
2 Inserisci il nodo radice nella lista
3 SE non ci sono più nodi da esplorare
    ALLORA fermati: la procedura ha fallito
4 SE non ci sono più nodi della lista
    ALLORA
4.1 Incrementa di 1 il valore di P
4.2 Ritorna al passo 2
5 SE il nodo all'inizio della lista è un nodo finale
    ALLORA fermati: hai risolto il problema
6 SE il nodo all'inizio della lista non ha figli OPPURE il nodo ha una profondità uguale a P

ALLORA toglilo dalla lista
ALTRIMENTI sostituisci tale nodo con i suoi figli inserendoli all'inizio della lista stessa
7 Ritorna al passo 3

Questa versione rappresenta il migliore algoritmo di ricerca cieca in profondità.

La ricerca in ampiezza si muove, anziché in profondità, esplorando i nodi per livelli successivi; prima di passare al livello N, analizza quindi tutti i nodi del livello N – 1. L'algoritmo può essere schematizzato come segue:.

Algoritmo di ricerca per ampiezza

0 Vuota la lista
1 Inserisci il nodo radice nella lista
2 SE non ci sono più nodi della lista
ALLORA fermati: la procedura ha fallito
3 SE il nodo all'inizio della lista è un nodo finale
ALLORA fermati: hai risolto il problema
4 SE il nodo all'inizio della lista non ha figli
ALLORA toglilo dalla lista
ALTRIMENTI sostituisci tale nodo con i suoi figli inserendoli alla fine della lista stessa
5 Ritorna al passo 2

Si confronti l'algoritmo con la prima versione dell'algoritmo di ricerca in profondità. Come si vede, essi sono identici, a meno dell'istruzione 4. Nel caso della ricerca in profondità, i nodi figlio devono essere inseriti all'**inizio**, nella ricerca in ampiezza alla **fine** della lista. La struttura dati utilizzata è, cioè, nel primo caso una **pila,** o **stack**, nel secondo una **coda**, o **queue**. Tali strutture di dati sono dette anche, rispettivamente, strutture di tipo LIFO (Last In First Out) e di tipo FIFO (First In First Out), e trovano un impiego assai diffuso nella rappresentazione dei dati in informatica. Vale la pena di notare come a volte la strutturazione dei dati rivesta un ruolo preponderante rispetto al comportamento di un algoritmo; troppo spesso si commette l'errore di attribuire molta enfasi allo sviluppo dei programmi, trascurando l'aspetto della strutturazione dei dati, o, più in generale, della rappresentazione della conoscenza, che viceversa gioca sistematicamente un ruolo preminente.

Abbiamo accennato al fatto che sia la ricerca in profondità che quella in ampiezza presentano comparativamente pregi e difetti. A favore della ricerca in ampiezza va ascritta la **robustezza**: essa, per come è concepita, non può andare in loop, né imboccare un sottoalbero infinito. Pertanto non è sensato introdurre routine di detezione di cicli o altri accorgimenti. Inoltre, nel caso di un problema che ammetta soluzioni multiple, essa troverà per prima la soluzione a minore distanza dalla radice, cioè, a parità di costo degli archi dell'albero, di costo minimo. D'altra parte, la ricerca in ampiezza presenta il difetto

che la rappresentazione dei dati in memoria ha crescita esponenziale: il passaggio da un livello a quello successivo fa aumentare il numero di nodi da rappresentare secondo il fattore di ramificazione. Viceversa, l'occupazione in memoria nella ricerca in profondità cresce in modo lineare. Questo per quanto riguarda il fattore spazio. Per quanto riguarda il fattore tempo, entrambe le procedure sono a crescita esponenziale. Si consideri il caso peggiore, ossia che non vi sia una soluzione, o, equivalentemente, che essa sia rappresentata dall'ultimo nodo a destra: entrambe percorreranno tutto l'albero.

La ricerca euristica

Finora ci siamo occupati di tecniche di ricerca "cieche", o altri dicono "non informate", o ancora basate sulla "forza bruta". Da un certo punto di vista si può dire che un approccio di questo genere rappresenta quanto di meno "intelligente" si può fare nella soluzione dei problemi. La ricerca cieca considera, infatti, tutti i nodi ugualmente promettenti. È tipico dell'intelligenza umana, viceversa, individuare percorsi più promettenti di altri. Ad esempio, il giocatore di scacchi non considererà tutte le mosse possibili da sinistra a destra, ma si concentrerà sul centro dell'azione, su un pezzo attaccato, su una minaccia Questo secondo approccio prende il nome di "ricerca euristica" (dal verbo greco *eurisko*, "trovo", reso celebre dal famoso "eureka", "ho trovato", di Archimede). Da un punto di vista matematico, un'euristica è una funzione che ha come dominio l'insieme dei nodi, e ad ognuno di essi attribuisce un valore, detto appunto "valore euristico". Normalmente, la funzione euristica stima il costo del nodo, e dunque un valore euristico sarà tanto migliore quanto più è basso. Introdurre una funzione di valutazione euristica ci consente di abbassare la complessità di un problema, riducendo considerevolmente lo spazio di ricerca. Ciò risulta di fondamentale importanza nei problemi a crescita esponenziale. Se un problema è a crescita esponenziale, infatti, dato che il lavoro che può essere svolto da un elaboratore cresce in modo uniforme nel tempo, vi sarà inevitabilmente un punto nel tempo oltre il quale il problema diventa insolubile. Mettendo la situazione in assi cartesiani, avremo

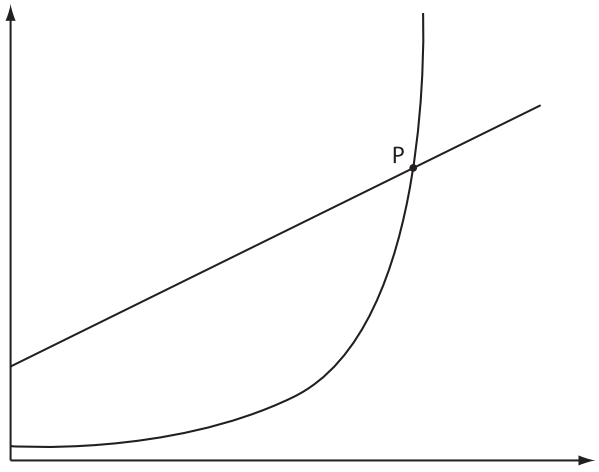

Figura 9

È chiaro che, oltre il punto P, il lavoro da svolgere per risolvere il problema cresce "di più" delle capacità del processore. Si noti che il discorso è del tutto teorico, e non dipende pertanto dall'hardware coinvolto: cambiare processore assumendone uno più potente non fa altro che spostare in avanti nel tempo il punto P, ma non ne esorcizza l'inevitabilità. L'assunzione di una euristica diventa quindi non tanto un'opportunità, quanto una vera e propria necessità logica.

L'assunzione di una euristica è un atteggiamento naturale dell'uomo. Si potrebbero fare centinaia di esempi tratti dalla vita di ogni giorno. Di fronte a una scelta complessa, o in vista di un obiettivo non direttamente raggiungibile, è del tutto normale, per parte nostra, dotarsi di una euristica. Così, quando dobbiamo scegliere un prodotto, in assenza di ulteriori informazioni, sceglieremo per esempio la marca più nota, o l'articolo più caro, o viceversa il più economico. Difficilmente invece sceglieremo in modo deterministico quello che sta più a sinistra. Ecco dunque che l'atteggiamento euristico è preso talvolta come sinonimo di "atteggiamento intelligente", in contrapposizione con la scelta casuale o con quella predeterminata da un criterio rigido. Ma proprio la varietà cui si fa riferimento vale a capire che la bontà di un'euristica non si dà come valore assoluto, ma è strettamente dipendente da molte circostanze, prima di tutto dal problema. Non esiste dunque un'euristica buona tout court, ma piuttosto una determinata euristica si dimostra più o meno adeguata ad un certo problema.

Gli algoritmi di ricerca euristica ricalcano quelli di ricerca cieca, con la sostanziale differenza che il passo successivo da compiere non viene scelto secondo un criterio predefinito, ma privilegiando il nodo che ha valore euristico migliore. Così abbiamo l'algoritmo così detto "hill climbing", che riproduce in forma euristica la ricerca in profondità, e l'algoritmo "best first", che si rifà a quella in ampiezza.

Hill climbing

La denominazione dell'algoritmo si ispira ad una metafora alpinistica. Si tratta di imitare il comportamento di uno scalatore che, cercando di raggiungere la vetta di una montagna, di fronte ad ogni bivio sceglie sempre il cammino che sale di più. L'algoritmo analizza dunque i figli del nodo attuale, e sceglie, per proseguire, il figlio di valore euristico migliore. Un algoritmo di questo genere è adeguato quando il problema è tale che la soluzione possa essere raggiunta in modo monotòno, cioè passando costantemente a nodi il cui valore euristico è migliore dei precedenti. Si dimostra invece debole di fronte a soluzioni sub-ottimali, o, in altre parole, ai così detti "massimi locali". Riprendendo la metafora alpinistica, il problema è quello delle vette secondarie: se si raggiunge una vetta che rappresenta una soluzione sub-ottimale, l'algoritmo non trova figli di valore euristico migliore della situazione attuale, e quindi termina. Pertanto, tutte le volte che il percorso per raggiungere la soluzione migliore presuppone un andamento non uniforme, ovvero che, nella solita metafora alpinistica, non è sempre in ascesa, l'algoritmo fallisce. Si possono introdurre tecniche per affrontare quest'ultimo problema. Ad esempio, far ripartire il computo in modo casuale e uscire così dalla vetta secondaria. L'hill climbing è considerato un algoritmo "tattico", o "locale". Il suo comportamento è molto simile a quello della ricerca cieca in profondità, perché si procede sempre dal padre ai figli, ma con l'essenziale differenza che

nella ricerca cieca il figlio viene individuato secondo un criterio prestabilito, mentre qui viene scelto in base alla funzione euristica.

Best first

L'algoritmo best first applica l'euristica in un modo diverso. Esso si chiede quale sia il nodo con il valore euristico migliore, indipendentemente dal fatto che sia figlio del nodo attuale. Il suo comportamento è dunque simile a una ricerca per ampiezza.

Algoritmo A

L'algoritmo A è una variante del *best first*. Anziché calcolare la stima di un nodo esclusivamente con la funzione euristica, l'algoritmo A valuta il costo di un cammino dalla radice a una foglia come la somma di due componenti, la parte del cammino già coperta, per la quale assume il costo effettivamente speso, e la parte residua, che viene stimata dall'euristica. In altri termini, supponiamo di chiederci il costo del cammino che va dalla radice A alla foglia Z, e di essere nel nodo N. Allora la stima sarà data dalla somma del costo da A a N, che è noto, e del costo da N a Z, che va valutato con l'euristica. Adesso chiediamoci che cosa succede al crescere di N, cioè passando da N a N+1. Avverrà che la parte certa della stima aumenta, mentre quella valutata dall'euristica diminuisce.

Un importante risultato legato all'algoritmo A è che si può dimostrare che, se l'euristica adottata è uniformemente ottimistica, esso trova la soluzione migliore, caratteristica che in gergo tecnico viene detta "ammissibilità". Un'euristica si dice uniformemente ottimistica quando, detto R(x) il costo reale di un'operazione che porta al nodo x, F(x) è sempre minore o uguale ad R(x). In altre parole, l'euristica non stima mai il costo maggiore del costo reale. In questo modo si capisce intuitivamente che non viene potato alcun cammino che potrebbe essere migliore di quelli analizzati. Date due euristiche uniformemente ottimistiche, si dice che l'euristica F^1 è più informata dell'euristica F^2 quando F^1 è costantemente maggiore di F^2. Così un'euristica più informata definirà uno spazio di ricerca che è un sottoalbero di quello generato dall'euristica meno informata. In altre parole, più un'euristica è informata e più riduce lo spazio di ricerca. All'opposto, l'euristica più ottimistica è quella che stima zero il costo di ogni nodo; essa è pertanto anche la meno informata, e di fatto si ritorna alla ricerca cieca: lo spazio di ricerca non viene diminuito affatto.

Problem solving con i grafi AND/OR

Una delle strategie più importanti nell'ambito delle tecniche di risoluzione dei problemi è quella di scomporre un problema in sottoproblemi più semplici. Si può dire che questo approccio è connaturato con il pensiero umano. Tanto per fare l'esempio di un precedente illustre, la seconda regola che Cartesio propone nel suo metodo è proprio l'analisi, cioè la scomposizione di un problema complesso in sottoproblemi più semplici. Per venire alla nostra disciplina, la strategia che stiamo considerando è già ben presente nell'ormai storico GPS di Simon e Newell. Ma esiste un modo

canonico per compiere una tale suddivisione? Non esiste certo una procedura meccanica, e il modo di procedere dipende ampiamente dallo specifico problema. Lo schema di ragionamento potrebbe essere sintetizzato così: ho un problema P; individuo i sottoproblemi SP1, SP2, SP3. Allora P è risolto quando sia stata trovata una soluzione per SP1, SP2, SP3. Tuttavia proprio questo modo di procedere suggerisce una considerazione naturale: non è detto che tutti i sottoproblemi siano legati da una congiunzione logica. Si può dare viceversa il caso che essi siano in alternativa. Supponiamo che il nostro problema sia di recarci a Roma. Allora si può ragionare così: si può andare a Roma in treno oppure in aereo. È chiaro che i due sottoproblemi che sono emersi da questa prima scomposizione non vanno soddisfatti entrambi, ma è sufficiente che ne sia soddisfatto uno solo. Per andare a Roma, ad esempio, in treno, si dovrà poi: avere soldi, comprare il biglietto, raggiungere la stazione, salire sul treno giusto. Questi altri sottoproblemi vanno soddisfatti tutti, pena il fallimento. Ecco dunque che dobbiamo affidarci ad un altro modo di rappresentare la situazione, diverso da quello di un grafo albero che esprima lo spazio degli stati. Qui abbiamo a che fare con i grafi AND/OR. Questo significa che i figli di un nodo possono essere congiunti, e allora vanno soddisfatti tutti, oppure disgiunti, e allora basta che uno solo sia soddisfatto.

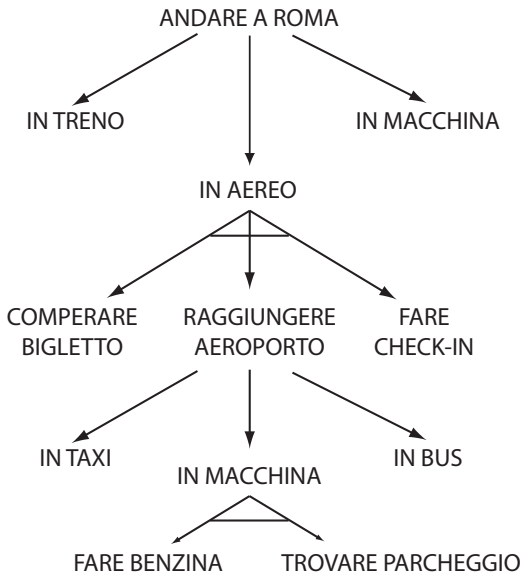

Figura 10 – Diversi modi per "andare a Roma".

Come si vede in figura, tracciamo una linea orizzontale a congiungere i figli dei nodi AND, mentre non mettiamo niente nel caso dei nodi OR.

Per determinare una soluzione in un grafo di questo tipo, dovremo far sì che per ogni nodo OR sia soddisfatto almeno un figlio, e per ogni nodo AND siano soddisfatti tutti. La soluzione non è più, dunque, un cammino dalla radice a una foglia voluta, come nei

casi visti in precedenza, ma è costituita da un sottoalbero, con le caratteristiche appena dette. Vediamo una soluzione del problema "andare a Roma"

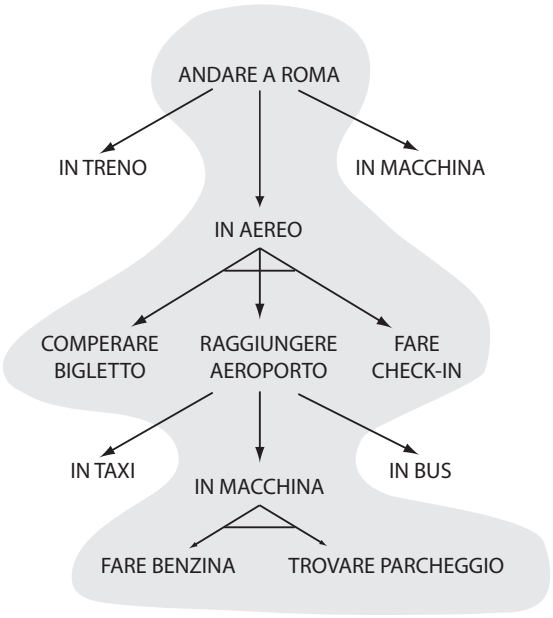

Figura 11 – Grafo AND/OR del problema "andare a Roma".

Come si calcola il costo di una soluzione in un grafo AND/OR? Non essendo un cammino, esso non può essere infatti assimilato alla somma dei costi degli archi. Facciamo l'ipotesi che la scomposizione del problema in sottoproblemi si arresti quando troviamo una "azione elementare", definita come un'azione direttamente eseguibile di cui è noto il costo. Allora avremo che, per ogni nodo del grafo,

➢ se è un'azione elementare il costo è dato per definizione

➢ se è un'azione elementare non eseguibile, allora il costo è infinito

➢ se è un nodo in OR assume il valore minimo tra quelli dei suoi figli

➢ se è un nodo AND allora vale la somma del valore dei figli.

In questo modo è possibile ribaltare all'indietro i costi del sottoalbero risolutivo, e computare il costo dei padri in dipendenza dei figli.

Il planning

Il planning è in un certo senso una evoluzione, o meglio una branca, del problem solving. Si tratta di assumere entro la panoramica della risoluzione dei problemi la dimen-

sione della temporalità, o, quanto meno, quella dell'ordine sequenziale, confrontandosi con la tematica di collegare tra di loro in una sequenza logica più problemi da risolvere. Non manca chi vede nel planning la principale caratteristica dell'intelligenza umana. Pianificare significa, in prima approssimazione, concepire l'esecuzione di una sequenza di azioni che conducano ad un risultato voluto.

Storicamente, i primi tentativi in questo campo furono fatti cercando di riadattare al nuovo contesto problemico la programmazione logica, e dunque, in definitiva, il calcolo dei predicati del primo ordine. Assumendo un tempo discreto, ossia come una successione di stati, si associa ad ogni predicato un ulteriore argomento, che specifica appunto a quale degli stati ci si riferisce. Così $P(a,s_1)$ significa che il predicato P vale di a allo stato s_1. In questo modo, secondo il consueto approccio della programmazione logica, lo stato finale voluto diventa il goal da dimostrare, e la catena delle successive unificazioni che si devono compiere per raggiungerlo esprime il "piano" voluto. È questo il così detto "calcolo situazionale" proposto da McCarthy e Hayes. Ma seguendo questa prospettiva insorgono notevoli difficoltà. La più rilevante di esse è che, in programmazione logica, non si può fare a meno, per ragioni di computabilità, della così detta "ipotesi del mondo chiuso", ossia, tutto ciò che non è esplicitamente assunto nel programma, o dedotto dalle premesse, è falso. Questo assunto consente la chiusura computazionale, ossia consente di rendere decidibile un calcolo, quello dei predicati appunto, che di per sé sarebbe "semi-decidibile". La conseguenza di questo assunto è che, essendo falso tutto ciò che non è esplicitamente asserito, si crea la necessità di riaffermare ogni volta, ad ogni passaggio di stato, anche ogni premessa che non è stata affetta dalle conseguenze di una azione. Questa circostanza prende il nome di "frame problem". Secondo il buon senso, ci pare ovvio che tutto ciò che non è stato cambiato rimanga tale e quale, e quindi che valgano nello stato successivo i predicati che valevano in quello precedente; tuttavia, per le ragioni tecniche anzidette, questa semplice caratteristica non si può assumere entro il prim'ordine.

Si sono così sviluppati, per affrontare le problematiche del planning, degli ambienti *ad hoc*, appositamente concepiti per la rappresentazione delle azioni nella loro temporalità. Il nuovo punto di vista si apre con il pianificatore STRIPS (*Standford Research Institut Problem Solver*). La novità essenziale è costituita da un nuovo modo di rappresentare le azioni. Anziché affidarsi al calcolo dei predicati, Fikes e Nilsson, gli autori di STRIPS, rappresentano un'azione come un insieme di *precondizioni* e un insieme di *conseguenze*. L'insieme delle precondizioni è una lista di enunciati che devono essere verificati perché l'azione possa avere luogo. L'insieme delle conseguenze è strutturato in due liste, quella delle *aggiunte* e quella delle *cancellazioni*. Questo approccio riesce pertanto a superare lo scoglio del frame problem, rappresentando soltanto i cambiamenti in modo esplicito. Un'azione in STRIPS avrà dunque la struttura:

Azione A

1. precondizioni: x, y, z
2. aggiunte: u, v
3. cancellazioni: k, w

dove x, y, etc. sono enunciati del calcolo dei predicati.

Questo approccio rappresenta una svolta nella pianificazione; STRIPS, malgrado i suoi limiti, di cui parleremo oltre, rimane per questo una pietra di paragone per i pianificatori successivi. L'algoritmo che governa l'andamento di STRIPS si basa sulla "analisi mezzi-fini", tecnica in larga misura derivante dal GPS. Essa consiste nel confrontare ricorsivamente lo stato attuale con quello voluto, nel ricercare un operatore che possa diminuire la differenza tra i due stati, e nell'assumere poi come nuovo stato voluto quello in cui si verificano le precondizioni dell'operatore che si vuole applicare in quanto appunto diminuisce la differenza rispetto allo stato finale. Il limite essenziale di STRIPS risulta essere quello che è un pianificatore "tattico", ossia che assume un obiettivo per volta, tentando di soddisfarli in sequenza. Il programma non prende quindi in alcuna considerazione il fatto che gli obiettivi possano interferire l'uno con l'altro, e debbano di conseguenza essere considerati in modo correlato. Più banalmente: operando su un obiettivo per volta, nulla vieta che, dopo il conseguimento di un obiettivo, l'attività svolta per raggiungerne un secondo di fatto infici il raggiungimento del primo. Infatti, se una delle azioni eseguite per raggiungere il secondo obiettivo ha nella lista delle cancellazioni un enunciato che compare nella lista delle aggiunte delle azioni che hanno realizzato il primo obiettivo, questo viene vanificato e distrutto. Avviene così che in situazioni anche non molto complesse STRIPS compia del lavoro inutile, o, ancora peggio, si immetta in un circolo vizioso, realizzando e poi distruggendo i suoi obiettivi.

I progressi successivi del planning hanno consentito di passare da pianificatori "tattici", come STRIPS appunto, a pianificatori "strategici", ossia in grado di trattare gli obiettivi congiuntamente. Le tecniche relative a questa evoluzione sono sostanzialmente tre: a) la protezione degli obiettivi; b) la regressione degli obiettivi; c) la modifica dei piani. La prima consiste nel "bloccare" le condizioni relative ad un obiettivo già raggiunto, in modo tale che esse diventino un vincolo per le azioni che vengono svolte successivamente. Così, se una condizione appartiene allo stato finale voluto, ed essa è stata già soddisfatta, non si potrà applicare un'azione che la contempli nella lista delle cancellazioni. La regressione degli obiettivi è in un certo senso una forma di backword-chaining; essa consiste nel calcolare a ritroso lo stato che precede lo stato finale. Se lo stato finale s_t dovrà soddisfare un certo obiettivo, e questo obiettivo è ottenuto con una certa azione A, allora è chiaro che nello stato immediatamente precedente, s_{t-1}, dovranno valere tutte le precondizioni di A. Infine, la terza tecnica, la modifica dei piani, consiste nel non dare per scontato che i vari sotto-piani debbano essere concatenati sequenzialmente. Supponiamo di avere ottenuto un primo obiettivo grazie al piano P_1. Supponiamo ora di elaborare un secondo piano, P_2, per raggiungere un secondo obiettivo. Il comportamento di STRIPS sarebbe quello di collocare P_2 in coda al primo. Ma in P_2 potrebbe comparire una qualche azione che danneggia il risultato di P_1. Ecco allora che, riscontrata questa situazione, il sistema tenta di collocare P_2 non in coda a P_1, ma allo stato precedente alla sua ultima azione. E così via ricorsivamente fino a trovare il punto corretto in cui P_2 può essere immerso in P_1 senza creare danni. **Si può dire che, ragionando in questo modo, emerge una regola del tutto generale: quando un'azione A danneggia un'azione B, ossia quando nella lista delle cancellazioni di A compare una condizione che appartiene alla lista delle aggiunte di B, allora si deve far sì che B sia eseguita in un momento che precede l'esecuzione di A**.

Un ulteriore sviluppo nella pianificazione discende dalla constatazione che il modo di pianificare dell'uomo è in grado di attribuire un'importanza diversa agli obiettivi, ossia di graduarne la criticità. In analogia con quanto avviene nel passaggio dalla ricerca cieca, in cui tutti i nodi sono equivalenti, e quella euristica, in cui i nodi vengono valutati secondo una funzione, si ha qui l'instaurarsi della pianificazione *gerarchica*. Questo significa, appunto, attribuire agli obiettivi un diverso *indice di criticalità*. L'algoritmo di planning cercherà dunque di soddisfare prima gli obiettivi di indice di criticalità maggiore. Si realizza così il classico modo di procedere top-down. Il vantaggio che si consegue secondo questo modo di procedere è che si evita di elaborare piani di dettaglio rispetto a un obiettivo di livello alto che non può essere raggiunto. Facciamo un esempio estremamente semplice. Supponiamo che io voglia produrre un piano per laurearmi in filosofia o in giurisprudenza. Se io procedessi sequenzialmente, secondo una programmazione non gerarchica, a questo punto dovrei andare a sviluppare i dettagli del primo obiettivo, ossia le azioni che lo rendono realizzabile. Avrei quindi "iscriversi a lettere e filosofia", "frequentare storia della filosofia", "frequentare…"; "leggere *La Metafisica* di Aristotele", "leggere… ", "sostenere l'esame di storia della filosofia", "sostenere l'esame di…", etc. E così via: ogni sotto-obiettivo deve ulteriormente essere decomposto, fino alle azioni elementari. Così "leggere *La Metafisica*" si svolge in OR in "comprare il libro" oppure "prenderlo a prestito in biblioteca" oppure "chiederlo in prestito a un amico". E così via ricorsivamente: "prenderlo a prestito" si svolge in "prenderlo a prestito nella biblioteca A", "prenderlo a prestito nella biblioteca B". Supponiamo infine che nella università della mia città, dove ho deciso di laurearmi, non esista la facoltà di lettere, ma solo quella di giurisprudenza. Ecco che tutto il piano sviluppato nei dettagli si rivela come un lavoro del tutto inutile. La gestione di un opportuno indice di criticalità eviterebbe tutto ciò, andando a verificare prima di tutto la soddisfacibilità degli obiettivi di più alto livello: è chiaro che, se non mi posso iscrivere, è del tutto inutile sviluppare i livelli più bassi del piano. L'approccio gerarchico consente quindi la realizzazione di piani sempre più di dettaglio, con una razionalità maggiore rispetto ai pianificatori lineari. Un esempio di riferimento per la pianificazione gerarchica è ABSTRIPS, sviluppato da Sacerdoti, che rappresenta una evoluzione di STRIPS con l'aggiunta, appunto, della gestione degli indici di criticalità.

Infine, prendiamo in considerazione un ulteriore miglioramento nel campo della pianificazione: la non linearità. L'esempio di riferimento è un pianificatore, anch'esso sviluppato da Sacerdoti come ABSTRIPS, che prende il nome di NOAH. L'idea di base può essere sintetizzata così. Il programma non deve produrre direttamente il piano definitivo, ma deve invece generare una successione di piani, ognuno dei quali è più raffinato del precedente. Un'altra idea fondativa di NOAH è il così detto principio del *least commitment*: il programma opera una scelta tra un'alternativa solo quando vi è costretto, il più tardi possibile. Il loop principale dell'algoritmo è dunque intuitivamente sintetizzabile come segue:

0 - elabora un piano

1 - critica il piano generato e miglioralo

2 - SE il risultato raggiunto è adeguato
ALLORA fermati
ALTRIMENTI torna al passo 1.

La tecnica che realizza il meccanismo del least commitment è quella di condurre in parallelo lo sviluppo di due azioni, fino a che non si è costretti a decidere quale delle due si deve fare prima dell'altra. Ciò si ottiene introducendo i nodi SPLIT e JOINT. Supponendo che debbano essere eseguite le azioni A e B. Anziché assumerle in sequenza, NOAH costruisce una struttura del tipo:

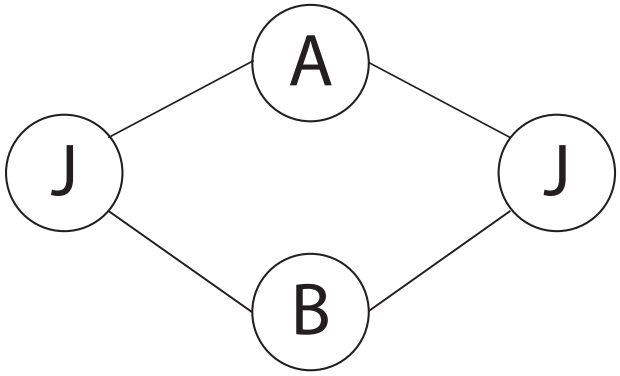

Figura 12

Nell'ambito dello sviluppo di un piano complesso, i miglioramenti successivi si otterranno, essenzialmente, dunque, con lo spostamento dei nodi SPLIT e JOINT. Il criterio di base per questo tipo di problemi è molto semplice, ed è sempre il solito in tutta la pianificazione: se un'azione A ha nella sua lista di cancellazioni un elemento che appartiene alla lista delle precondizioni di una seconda azione B, si dice che "A danneggia B" in quanto la rende irrealizzabile. Da ciò segue l'ovvia conclusione che l'azione danneggiata deve comunque essere eseguita **prima** dell'azione danneggiante.

Nella presentazione informale del loop principale sopra abbiamo introdotto l'espressione: "critica il piano". La nozione appare in prima battuta assai vaga. In realtà, essa rimanda invece ad una precisa strategia algoritmica, e cioè l'invocazione da parte del programma di una serie di sub-routine che vengono appunto dette "critici". I critici sono dunque sottoprogrammi il cui scopo è quello di valutare la bontà di un piano, rilevarne le incongruenze, e modificarlo di conseguenza, migliorandolo. Cominciamo ad esemplificare dal caso più semplice, il "critico delle ridondanze". Supponiamo che la precondizione P sia presente tanto nella lista dell'azione A quanto nella lista dell'azione B, che sono entrambe da compiere. Allora è evidente che essa è ridondante, ovvero è inutile che nel piano definitivo compaia due volte, posto che deve essere soddisfatta prima dell'esecuzione della prima delle due azioni considerate.

Il più importante dei critici è senz'altro quello che rileva i conflitti, ossia individua quali siano le azioni danneggianti e quali quelle danneggiate. Questo critico, una volta riscontrata tale anomalia, sposterà i nodi SPLIT-JOINT, in modo che l'azione danneggiante debba necessariamente essere eseguita **dopo** l'azione danneggiata.

È importante notare che, se non vi sono conflitti, viene mantenuta la struttura in parallelo dello sviluppo del piano. Un pianificatore non lineare come NOAH può dunque generare un piano finale che lascia alcuni segmenti dello stesso in parallelo, il che va interpretato come il fatto che è indifferente eseguire, ad un certo punto, prima A e poi B o prima B e poi A.

Bibliografia

Per approfondire queste tematiche e per una bibliografia cartacea, ben documentata ed annotata, rimando al seguente volume:

Fum D. (1994), *Intelligenza artificiale*, Bologna, Il Mulino.

Ulteriori e più recenti testi di riferimento sono:

Nilsson N. J. (1998), *Artificial intelligence: a new synthesis*, San Francisco, Morgan Kaufmann, (trad. it., *Intelligenza artificiale*, Milano, Apogeo, 2002).

Poole D., Mackworth A., Goebel R. (1998), *Computational intelligence: a logical approach*, Oxford, Oxford University Press.

Russell S., Norvig P. (2003), *Artificial intelligence: a modern approach*, 2nd ed., Upper Saddle River, NJ, Prentice Hall/Pearson Education (trad. it., *Intelligenza artificiale: un approccio moderno*, 2ª ed., Milano, Pearson Education Italia, 2005).

È indubbio però che, data la modernità e la dinamicità degli argomenti qui affrontati, la cosa migliore è ricercare sulle numerosissime informazioni on line. Riporto di seguito l'indirizzo di alcuni siti tra i più interessanti e completi, a mio modo di vedere, di manuali on line o bibliografie; molti di essi, a loro volta, rimandano ad altri siti, attraverso un'infinità di link.

http://www.ing.unife.it/informatica/AppliIA/testi.shtml

http://www.di.unipi.it/~simi/AI/SI2005/lucidi.html

http://aima.cs.berkeley.edu/

http://citeseer.ist.psu.edu/articles.html

http://liinwww.ira.uka.de/bibliography/Ai/

http://liinwww.ira.uka.de/bibliography/Ai/others.html

http://consc.net/biblio/4.html

http://users.ox.ac.uk/~econec/cogsciai.html

http://bubl.ac.uk/link/a/artificialintelligence.htm