

UNIVERSITÀ DEGLI STUDI DI VERONA

Dispensa del corso di Intelligenza Artificiale

Sebastiano Fregnan

9 novembre 2022

Indice

1	Agenti razionali	5
1.1	Definizioni	5
1.2	Task environment	6
1.3	Architetture degli agenti	6
1.4	Modelli dell'ambiente	7
1.5	Categorie di agenti goal-based	7
1.6	Categorie di problemi	8
1.7	Approcci ai problemi	8
2	Problemi di ricerca	9
2.1	Definizione del problema	9
2.2	Problema - Lupo, Pecora e Cavolo: definizione	9
2.3	Ricerca della soluzione	10
2.4	Strategie di ricerca	11
2.5	Strategie uninformed	11
2.5.1	Breadth-first search	11
2.5.2	Uniform-cost search	11
2.5.3	Depth-first search	11
2.5.4	Depth-limited search	12
2.5.5	Iterative deepening search	12
2.5.6	Comparazione strategie uninformed	12
2.6	Strategie informed	13
2.6.1	Greedy Best-First search	13
2.6.2	A* search	14
2.7	Problema - Lupo, Pecora e Cavolo: ricerca	16
3	Problemi di ottimizzazione	17
3.1	Strategie locali	17
3.1.1	Hill-Climbing (o gradient descent)	17
3.1.2	Simulated Annealing	17
3.1.3	Local Beam Search	18
3.1.4	Algoritmi genetici	18
3.1.5	Ricerca locale in spazi continui	19
3.2	Problema - 4 Regine	20
4	Constraint Processing	21
4.1	Reti a vincoli	21
4.1.1	Rappresentazione dei vincoli	22
4.1.2	Tecniche di soluzione	23
4.2	Propagazione dei vincoli	23
4.3	Algoritmi di propagazione	24
4.4	Tecniche di ricerca	26
4.5	Ordinamento delle variabili	27
4.6	Considerazioni di costo	29
4.7	Look-ahead	30
4.7.1	Forward Checking look-ahead	30
4.7.2	Arc Consistency look-ahead	30
4.8	Ristrutturazione del problema	31
4.8.1	Cycle conditioning	31
4.8.1.1	Tree decomposition	32
4.8.1.2	Dual Based Recognition	35
4.8.1.3	Primal Based Recognition	36
4.8.2	Clustering	38
4.8.2.1	Join Tree Clustering	39

5	Constraint Optimization	43
5.1	Reti di costo	43
5.2	Tecniche di soluzione	44
5.3	Branch and Bound	44
5.4	Bucket Elimination	46
5.4.1	Trasformazione degli Hard Constraint	48
5.4.2	Inclusione degli Hard Constraint	48
6	Incertezza	51
6.1	Basi	51
6.1.1	Inferenza per enumerazione	53
6.1.2	Regola di Bayes	54
6.1.3	Problema - Mondo del Wumpus	55
6.2	Sequential Decision Process	56
6.2.1	Sequenze di decisioni	56
6.2.2	Transition Tree	57
6.2.3	Policy	57
6.2.4	Utility	58
6.2.5	Alberi di decisione	59
6.3	Markov Decision Process	60
6.3.1	Risolvere un MDP	61
6.3.1.1	Policy Iteration	62
6.3.1.2	Value Iteration	63
6.4	Osservabilità parziale	63
7	Learning	65
7.1	Reinforcement Learning	65
7.1.1	Approccio model-based	66
7.1.2	Approccio model-free	66
7.1.2.1	Temporal Difference Learning	67
7.1.2.2	Q-Learning	68
7.1.2.3	SARSA	69
7.1.3	Dilemma Exploration vs Exploitation	70
7.1.4	Conclusioni	70
7.2	Deep Reinforcement Learning	70
7.2.1	Feed-Forward Artificial Neural Network	71
7.2.2	Deep Neural Networks	71
7.2.3	Vanishing Gradient	71
7.2.4	Deep Q-Learning	72
7.2.5	Conclusioni	73

1 | Agenti razionali

1.1 Definizioni

Un agente è un'entità che percepisce eventi e agisce di conseguenza, come un umano, un robot o un termostato. Astrattamente, un **agente razionale** è rappresentato da una funzione (detta **funzione agente**) che mappa l'intero storico di **percezioni** possibili, input ottenuti dall'ambiente tramite **sensori**, a delle **azioni**, output compiuti tramite degli **attuatori**, formalmente

$$f : \mathcal{P}^* \rightarrow \mathcal{A}.$$

Il **programma agente** che implementa tale agente viene eseguito su una **architettura** fisica, per riprodurre alcuni comportamenti della **funzione agente**.

Esempio

Un Roomba tramite vari sensori percepisce il luogo in cui si trova e se lì è presente o meno sporcizia, e per agire di conseguenza può decidere di andare a destra, a sinistra, di aspirare o di non fare nulla. Le sue percezioni sono quindi tuple nella forma $\mathcal{P}_i = [location, status]$ e le sue azioni sono nell'insieme $\mathcal{A} = \{Suck, Left, Right, NoOp\}$; l'insieme di tutte le percezioni possibili è

$$\mathcal{P} = \{[A, Dirty], [A, Clean], [B, Dirty], [B, Clean]\}.$$

Notiamo che dato un agente con $|\mathcal{P}|$ possibili percezioni, la funzione agente a tempo T avrà $\sum_{t=1}^T |\mathcal{P}|^t$ possibili comportamenti: il goal dell'intelligenza artificiale è progettare programmi agente *piccoli* per rappresentare funzioni agente infinite, come ad esempio

Algorithm 1 Programma agente per l'aspirapolvere

```
1: function REFLEX-VACUUM-AGENT(location,status)
2:   if status is Dirty then return Suck
3:   else if location is A then return Right
4:   else if location is B then return Left
```

Per valutare un programma agente vengono usate delle **performance measure** che osservano la sequenza di percezioni e premiano o meno l'agente (es. un punto per ogni luogo pulito in tempo T , oppure un punto per ogni luogo pulito in un time step meno uno per ogni movimento, ...). Un agente cercherà quindi di massimizzare la **expected value** (valore atteso) della performance measure data la sequenza di percezioni. Attenzione però, "razionale" non significa onniscente né tantomeno chiaroveggente (infatti massimizziamo il valore *atteso*, non il valore *effettivo*), quindi di conseguenza nemmeno di successo, mentre invece è sinonimo di esplorazione, apprendimento e autonomia. Se rappresentiamo ogni possibile azione intrapresa dall'agente con un albero, il punteggio massimo è ottenibile da uno o più percorsi che massimizzano le condizioni della performance measure.

Ronda multi-robot

Impostiamo un ambiente di 3 stanze A, B, C e due robot r_1, r_2 , dove r_1 è di ronda in A e B mentre r_2 in B e C , trovandosi inizialmente rispettivamente in A e C . Il tempo di transizione da una stanza all'altra è 0 e la performance measure è minimizzare la somma della Idleness media di tutte le stanze, cioè del totale degli intervalli di tempo nei quali una stanza non contiene un robot rispetto al tempo totale. Un possibile approccio potrebbe essere tenere fermo un robot e far muovere solo l'altro; un secondo approccio invece potrebbe essere switchare i robot a tempi alterni.

Algorithm 2 Approccio “fix”

```

1: procedure FIXAPPROACH
2:   loop
3:     MOVEROBOT( $r_1$ )
4:     CLEAN( $r_1$ )

```

Algorithm 3 Approccio “switch”

```

1: procedure SWITCHAPPROACH
2:   CurrentRobot  $\leftarrow r_2$ 
3:   loop
4:     if time % 2 is 0 then
5:       CurrentRobot  $\leftarrow r_1$ 
6:     else
7:       CurrentRobot  $\leftarrow r_2$ 
8:     MOVEROBOT(CurrentRobot)
9:     CLEAN(CurrentRobot)

```

Questo secondo algoritmo ha la stessa expected value del primo, $1/3$, ma se osserviamo la standard deviation notiamo che è nettamente maggiore rispetto al primo, quindi forse sarebbe meglio cambiare la performance measure: notiamo infatti che anche mantenendo sempre i robot fermi abbiamo la stessa expected value. Un altro tweak sarebbe considerare invece la maximum Idleness.

1.2 Task environment

Per progettare un agente razionale dobbiamo quindi specificare:

Performance measure in che modo viene valutato l’operato dell’agente;

Environment quali caratteristiche circondano l’agente;

Actuators come l’agente si interfaccia con l’ambiente per compiere le azioni;

Sensors attraverso cosa l’agente riceve informazioni sull’ambiente.

Queste caratteristiche specificano il *task environment* (o anche *PEAS*), ovvero la situazione nella quale si trova l’agente. I diversi task environment possono essere raggruppati in diverse categorie in base alle loro proprietà:

Completamente o Parzialmente osservabile se il mondo è totalmente noto o meno. Un terzo caso è il **non osservabile**, dove l’agente non ha sensori per ricevere informazioni dall’ambiente;

Single- o Multi-agent se nell’ambiente ci sono altri agenti, con i loro specifici compiti. Nei casi multi-agent si avranno comportamenti **cooperativi** oppure **competitivi**;

Deterministico o Stocastico se le l’ambiente a seguito delle azioni compiute ha o meno componenti aleatorie. Un terzo caso è il **nondeterministico**, dove le azioni possono produrre diverse nuove configurazioni dell’ambiente ma senza sapere con quali probabilità;

Episodico o Sequenziale se ogni azione è indipendente dall’azione precedente;

Statico o dinamico se un ambiente può cambiare in modo indipendente nel tempo. Un terzo caso è il **semidinamico**, nel quale non è l’ambiente a cambiare ma il punteggio di performance ottenuto dall’agente;

Discreto o continuo se l’ambiente può essere visto come insieme di stati distinti;

Noto o ignoto se l’agente conosce o meno come funziona l’ambiente che lo circonda.

Ovviamente il caso peggiore è un task environment parzialmente osservabile, multi-agent, stocastico, sequenziale, dinamico, continuo e ignoto, che è praticamente il mondo in cui si trova un neonato.

1.3 Architetture degli agenti

La progettazione di un agente quindi è fortemente dipendente dalle proprietà del suo task environment, e diciamo che in generale possiamo avere quattro tipi di architettura degli agenti, in ordine di generalità crescente:

Simple reflex agents reagisce di conseguenza all’input (in pratica se riceve una certa percezione allora agisce con una certa azione);

Model-based reflex agent o *model agents* reagisce all'input in base al suo stato interno (ed è quindi un *simple reflex* con memoria delle percezioni precedenti);

Goal-based agents reagisce in base a ciò che le sue azioni producono nello stato corrente per raggiungere un *goal* finale nel futuro, in pratica agisce con *cognizione di causa* (ed è quindi un *model-based* che proietta le conseguenze delle sue azioni);

Utility-based agents reagisce in base al suo stato corrente con una specifica azione se decide che lo stato successivo migliorerà il suo stato attuale (ed è quindi un *goal-based* che cerca la *felicità*).

Ognuno di questi tipi di agenti può essere rimodellato come un *learning agent*, che però ha un'architettura totalmente differente dai precedenti, infatti il suo operato viene valutato, mediante l'assegnamento di *reward* o *penalty*, in base alle sue performance complessive e l'agente cambia il suo approccio al problema di volta in volta per migliorare tali performance.

1.4 Modelli dell'ambiente

Le componenti interne di un agente, ovvero i modelli matematici che devono rappresentare la *cognizione di causa*, la *felicità* o anche semplicemente la *memoria*, sono distribuiti in tre categorie su un asse di espressività crescente:

Rappresentazione atomica ogni stato del mondo è indivisibile e ogni stato è a se stante, non ha nulla in comune con gli altri stati (ad esempio, il mondo è visto semplicemente come la città in cui mi trovo). Algoritmi di *ricerca*, gli *Hidden Markov Models* e i *Markov Decision Processes* interpretano l'ambiente come se fosse atomico;

Rappresentazione composta (factored) ogni stato del mondo ha delle variabili interne che ne determinano un certo grado di similarità con gli altri stati (ad esempio, il mondo è visto come il livello di benzina corrente, le spie luminose del cruscotto accese, la città in cui mi trovo, la stazione che suona alla radio, i soldi nel portafoglio). La *logica proposizionale*, la *pianificazione* e le *reti Bayesiane* utilizzano rappresentazioni composte;

Rappresentazione strutturata ogni stato del mondo è composto di entità in relazione tra loro (ad esempio, il mondo è visto come l'auto che si trova sulla strada, io che mi trovo nell'auto, l'auto che viaggia ad una certa velocità, il serbatoio che contiene un certo livello di benzina). I *database relazionali*, la *logica del primo ordine* ed il *linguaggio naturale* utilizzano rappresentazioni strutturate.

Ogni tipo di rappresentazione ingloba quelle meno espressive, ad esempio un database relazionale (rappr. strutturata) può contenere un'entità che possiede un solo attributo (rappr. composta) il quale rappresenta la città in cui mi trovo (rappr. atomica). In base alla necessità di esprimere i dettagli dell'ambiente viene scelta, in fase di progettazione del programma agente, una rappresentazione piuttosto che un'altra, tenendo conto del fatto che più è alta l'espressività più sarà difficile lo sviluppo del programma, o l'apprendimento automatico: un sistema intelligente infatti dovrebbe essere capace di operare usando tutti i livelli di espressività per trarre vantaggi assoluti, obiettivo ovviamente arduo e spesso non raggiungibile.

1.5 Categorie di agenti goal-based

Un *reflex agent* decide le sue azioni in base allo stato dell'ambiente e dunque un ambiente troppo grande renderebbe impossibile mappare ogni stato ad una azione; d'altro canto, un *model-based agent* mappa gli stati solo se hanno una relazione con il proprio stato interno, e quindi hanno una limitazione per quanto riguarda il raggiungimento di un obiettivo in generale. Poniamoci quindi nel caso di un *goal-based agent*, il quale è fatto per raggiungere uno specifico goal senza gestire singolarmente ogni stato dell'ambiente. In base alla rappresentazione del mondo che decidiamo di usare possiamo categorizzare questi specifici agenti in due tipi:

Problem-solving agents se usiamo una *rappresentazione atomica*. L'agente avrà come unico scopo il raggiungimento ottimale del suo obiettivo, senza considerare altre variabili presenti nel mondo (proprio perché utilizza una rappresentazione atomica);

Planning agents se usiamo invece una *rappresentazione composta* o *strutturata*. In questo caso l'agente dovrà tenere conto di tutte le variabili che compongono il mondo e dovrà quindi pianificare una strategia, ottimizzando costi e benefici al variare delle soluzioni possibili (ad esempio percorrere 100km in una direzione potrebbe esaurire tutto il carburante nel serbatoio e quindi sarebbe necessario percorrere una strada con un distributore a meno di 100km).

1.6 Categorie di problemi

In base all'ambiente in cui ci troviamo andremo incontro a diversi tipi di problemi, che quindi categorizziamo in base alle proprietà del *task environment*:

Single-state problem se il PEAS è *deterministico* e *completamente osservabile*: la soluzione è unica ed è una *sequenza* di azioni. L'agente non avrà mai bisogno di osservare le percezioni più di una volta poiché conosce a priori l'intero ambiente;

Conformant problem se il PEAS è *deterministico* ma *parzialmente osservabile*: la soluzione potrebbe non esserci, ma se esiste allora è una *sequenza*. L'agente potrebbe non avere idea di dove si trovi o comunque ha una visione ristretta dell'ambiente, sa cosa vuole raggiungere ma non ha idea di dove esso si trovi rispetto a se, quindi il massimo che può fare è ipotizzare com'è fatta la porzione di ambiente non osservabile e proporre una soluzione, che però potrebbe non essere di successo. In questo caso, l'agente potrebbe aver bisogno di ulteriori informazioni dalle percezioni, per riformulare la soluzione al problema;

Contingency problem se il PEAS è *nondeterministico* e/o *parzialmente osservabile*: la soluzione è una *policy*, che mappa uno stato dell'ambiente a delle azioni. L'agente non può ignorare le percezioni, poiché esse danno nuove informazioni assolutamente necessarie per il raggiungimento dell'obiettivo e quindi esegue le stesse azioni fino al ricevimento di nuove percezioni (per esempio, mantengo i 50km/h finché non trovo il cartello che obbliga ad andare a 30km/h);

Exploration problem se il PEAS è *ignoto*: la soluzione non può essere trovata senza considerare le percezioni, poiché solo tramite esse è possibile determinare le regole che governano l'ambiente (sarà necessario un approccio *online*).

1.7 Approcci ai problemi

Durante l'implementazione del programma agente, il progettista deve decidere come elaborare il piano di risoluzione del problema e in particolare se creare la soluzione di volta in volta, in base alle percezioni ricevute, o se invece crearla completamente all'inizio per poi eseguire le azioni decise passo passo. Individuiamo quindi due approcci al *problem-solving*:

Offline problem-solving l'agente utilizza la prima percezione (ovvero i dati iniziali dell'ambiente) per analizzare il problema e produce una sequenza di azioni da eseguire imperativamente;

Online problem-solving l'agente crea una soluzione iniziale del problema e poi la rielabora continuamente in base alle percezioni che riceve, cambiando di volta in volta la sequenza di azioni da compiere.

Notiamo che in generale l'approccio *online* è adoperabile in tutte le categorie di problemi viste sopra, ma è ovviamente più oneroso computazionalmente, gestendo ed elaborando tutte le percezioni raccolte. Al contrario invece, l'approccio *offline* è leggero (viene calcolato tutto all'inizio) ma è adoperabile con garanzia di successo solamente per i *single-state problem*, poiché in essi non si devono considerare i problemi legati al nondeterminismo o alla parziale osservabilità dell'ambiente e quindi dato il piano di esecuzione si ha la certezza di raggiungere l'obiettivo richiesto.

Ad esempio, immaginiamo di voler andare da Verona a Roma e che il tratto autostradale passante per Firenze sia chiuso per lavori in corso. Se avessimo *osservabilità totale* dell'ambiente il problema sarebbe gestibile come *single-state problem* e quindi potremmo adoperare un agente *offline*, poiché durante il calcolo iniziale del tragitto eviterebbe di prendere l'autostrada in quel tratto, uscendo al casello prima. In caso di *osservabilità parziale* invece dovremmo usare un agente *online*, che avrebbe valutato l'efficienza del prendere l'autostrada e, *raggiunto* il tratto di Firenze, avrebbe percepito che il tratto era chiuso (tramite sensori che leggono cartelli stradali ad esempio) e sarebbe uscito al casello precedente, cambiando strada, ma mantenendo il suo goal, andare a Roma.

2 | Problemi di ricerca

Concentriamoci ora sui problemi più semplici da affrontare, ovvero i *single-state problem* e, data la categoria del problema, adoperiamo quindi un *problem-solving agent* con approccio *offline*. Dunque, dato l'ambiente e la prima percezione, l'agente dovrà:

1. *formulare il goal*, quale stato finale dell'ambiente si desidera raggiungere;
2. *formulare il problema*, come rappresentare il problema in termini di stati dell'ambiente e di azioni;
3. *individuare la soluzione*, come raggiungere il goal usando gli stati e le azioni individuate prima.

Possiamo vedere l'implementazione dell'agente nell'algoritmo:

Algorithm 4 Problem-solving agent

```
1: static actionSeq  $\leftarrow$  [], currState, goal, problem  $\leftarrow$  null
2: function PROBLEMSOLVINGAGENT(percept)
3:   currState  $\leftarrow$  UPDATESTATE(currState, percept)
4:   // initiate using first perception
5:   if actionSeq is empty then
6:     goal  $\leftarrow$  FORMULATEGOAL(currState)
7:     problem  $\leftarrow$  FORMULATEPROBLEM(currState, goal)
8:     actionSeq  $\leftarrow$  SEARCH(problem)
9:     if actionSeq is failure then return NoOp
10:  action  $\leftarrow$  FIRST(actionSeq)
11:  actionSeq  $\leftarrow$  REST(actionSeq)
12:  return action
13: loop
14:  perception  $\leftarrow$  GETPERCEPTION(sensors)
15:  PROBLEMSOLVINGAGENT(perception)
```

Si noti che le varie funzioni sono generiche e vanno approfondite in base allo specifico problema, a parte l'algoritmo di ricerca, che può essere generalizzato.

2.1 Definizione del problema

La formulazione di un problema si può ridurre a cinque punti fondamentali:

Stato iniziale dell'ambiente in cui si trova l'agente inizialmente;

Azioni che l'agente può compiere;

Transition model che descrive il risultato di ogni azione per ogni stato dell'ambiente;

Goal test che valuta se è stato raggiunto il goal dell'agente. A volte può essere semplicemente un'insieme di stati desiderati (ad esempio, essere a Roma), altre volte può essere una descrizione di una proprietà astratta (ad esempio, lo scacco matto);

Path cost costo di ogni decisione presa, ovvero il prezzo per passare da uno stato ad un altro tramite una specifica azione (ad esempio, il numero di chilometri da Verona a Modena, dopo aver deciso di compiere l'azione "andare a Modena" essendo a Verona).

2.2 Problema - Lupo, Pecora e Cavolo: definizione

Usiamo quanto visto sopra per definire un primo problema giocattolo: un contadino M deve trasportare da sinistra a destra di un fiume un lupo W , una pecora S e un cavolo C , usando una barca B . Lupo e pecora non

possono rimanere insieme sulla stessa sponda, così come pecora e cavolo e il contadino può trasportare una cosa alla volta.

Innanzitutto dobbiamo costruire il modello del problema e decidiamo di rappresentare l'ambiente come una tupla (W, S, C, M, B) , in cui le variabili prendono valore 0 se sono a sinistra oppure 1 se sono a destra. Lo stato iniziale è quindi $(0, 0, 0, 0, 0)$ e quello di goal è $(1, 1, 1, 1, 1)$. Dobbiamo ora definire le azioni, considerando le precondizioni necessarie per la loro esecuzione:

$CARRYWOLF (W, S, C, M, B) \rightarrow (\overline{W}, S, C, \overline{M}, \overline{B})$ se $S \neq C, W = M, M = B$;

$CARRYSHEEP (W, S, C, M, B) \rightarrow (W, \overline{S}, C, \overline{M}, \overline{B})$ se $S = M, M = B$;

$CARRYCABBAGE (W, S, C, M, B) \rightarrow (W, S, \overline{C}, \overline{M}, \overline{B})$ se $W \neq S, C = M, M = B$;

$CARRYNOTHING (W, S, C, M, B) \rightarrow (W, S, C, \overline{M}, \overline{B})$ se $W \neq S, S \neq C, M = B$;

Notiamo che ogni precondizione contiene il termine $M = B$, ovvero che il contadino deve essere assieme alla barca, e quindi considerare B singolarmente è del tutto inutile e ridondante: possiamo eliminarlo dalla rappresentazione dello stato e alleggerire quindi il modello, utilizzando la tupla (W, S, C, M) . Per verificare se il goal è raggiunto è necessario semplicemente un confronto booleano sullo stato attuale, mentre per il path cost possiamo assumere un valore costante.

2.3 Ricerca della soluzione

Per cercare la soluzione al problema l'agente può utilizzare degli *algoritmi di ricerca*, che trovano all'interno dell'albero di stati espansi del sistema il percorso più efficiente per massimizzare la *expected value*. Utilizziamo quindi inizialmente un algoritmo di *tree-search*, il quale esplora offline lo spazio degli stati generando i successori degli stati già esplorati (*espandendo* il sottoalbero di ogni stato).

Algorithm 5 Tree-search algorithm

```

1: function TREESearch(problem, frontier)
2:   root  $\leftarrow$  MAKENode(problem.InitialState)
3:   INSERT(root, frontier)
4:   loop
5:     if frontier is empty then return failure
6:     node  $\leftarrow$  REMOVEFRONT(border)
7:     if problem.GOALTEST(node.State) then return node
8:     expanded  $\leftarrow$  EXPAND(node, problem)
9:     INSERTALL(expanded, frontier)
```

Ovviamente espandendo è possibile tornare allo stato da cui veniamo, in quanto l'espansione considera anche il caso in cui viene compiuta l'azione inversa a quella compiuta per arrivare a quel nodo espanso, con il rischio di espandere all'infinito l'albero degli stati. Per evitare questo problema quindi non utilizziamo un albero bensì un grafo di stati e descriviamo di conseguenza un algoritmo di *graph-search*.

Algorithm 6 Graph-search algorithm

```

1: function GRAPHSEARCH(problem, frontier)
2:   explored  $\leftarrow$  [ ]
3:   root  $\leftarrow$  MAKENode(problem.InitialState)
4:   INSERT(root, frontier)
5:   loop
6:     if frontier is empty then return failure
7:     node  $\leftarrow$  REMOVEFRONT(frontier)
8:     if problem.GOALTEST(node.State) then return node
9:     if node.State not in explored then
10:       INSERT(node.State, explored)
11:       expanded  $\leftarrow$  EXPAND(node, problem)
12:       INSERTALL(expanded, frontier)
```

Gli algoritmi di tree- e graph-search sono molteplici e sono identificati dalla *strategia* con cui aggiungiamo e ordiniamo i nodi nella frontiera, ovvero della lista dei nodi ancora da esplorare; di seguito li vedremo in dettaglio.

2.4 Strategie di ricerca

La strategia di un algoritmo di ricerca è definita in base all'ordine in cui poniamo i nodi espansi all'interno della frontiera. Le strategie vengono valutate su quattro dimensioni:

Completezza se la soluzione viene sempre trovata, quando esiste;

Ottimalità se la soluzione è sempre quella meno costosa;

Complessità temporale e spaziale in termini di nodi generati/espansi e in memoria, misurata mediante i parametri:

Branching factor b diramazione massima dei nodi dell'albero;

Least-cost depth d profondità del percorso meno costoso;

Maximum depth m profondità massima dello spazio degli stati;

Least-cost solution cost C^* costo della soluzione con costo minimo.

In base alle informazioni disponibili dalla definizione del problema, in particolare alla relazione tra stato attuale e stato goal, le strategie possono essere *informed* o *uninformed*. Inoltre, definiamo la funzione *step cost* $c(s_i, a_j, s_{i+1})$, che indica quanto costa passare dallo stato s_i allo stato s_{i+1} tramite l'azione a_j .

2.5 Strategie uninformed

Utilizziamo queste strategie quando non sappiamo nulla della relazione tra lo stato attuale e quello goal. Le tecniche di ricerca usate principalmente sono le seguenti.

2.5.1 Breadth-first search

La frontiera è una coda FIFO, quindi si procede in ampiezza sulle espansioni dei nodi. Garantisce sempre il raggiungimento dello stato goal tramite il percorso a costo minimo, ma non garantisce che il raggiungimento sia efficiente:

Algorithm 7 BFS

```

1: function BREATHFIRSTSEARCH(problem)
2:   node  $\leftarrow$  MAKENODE(problem.InitialState)
3:   if problem.GOALTEST(node.State) then return node
4:   frontier  $\leftarrow$  NEWFIFO(node)
5:   explored  $\leftarrow$  {}
6:   loop
7:     if frontier is empty then return failure
       // usa il nodo più in superficie
8:     node  $\leftarrow$  frontier.POP
9:     add node.STATE to explored
10:    for all action in problem.ACTIONS(node.State) do
11:      child  $\leftarrow$  CHILD-NODE(problem,node,action)
12:      if child.State is not in explored or frontier then
13:        if problem.GOALTEST(child.State) then return child
14:        frontier.INSERT(child)

```

2.5.2 Uniform-cost search

La frontiera è ordinata per costo minimo del cammino, è equivalente alla BFS se tutti gli step hanno costo uguale. **CODICE**

2.5.3 Depth-first search

La frontiera è una coda LIFO, quindi si procede in profondità sulle espansioni dei nodi. Questa strategia in tree-search non è completa, infatti potrebbe non terminare se sono presenti cicli nell'albero (cioè se abbiamo un grafo), e dobbiamo usarla quindi in un algoritmo graph-search. **CODICE**

2.5.4 Depth-limited search

Ibrido tra BFS e DFS, nel quale si procede in profondità fino ad un certo limite ℓ detto **cutoff**, per poi proseguire in ampiezza e ricominciare. **CODICE**

2.5.5 Iterative deepening search

Si tratta letteralmente di una DLP con **cutoff** che cresce fino a che non viene trovata la soluzione. Può essere modificata per seguire il percorso meno costoso, come fa la UCS, ed è per questo la strategia più usata (riesce a trarre i pregi di tutte le strategie viste). **CODICE**

2.5.6 Comparazione strategie uninformed

Passiamo ora alle performance, mettendo per iscritto che nel calcolo delle complessità la radice non viene contata perché effettivamente non viene generata da nessuna espansione. Abbiamo quindi la tabella:

Strategia	Completo	Ottimale	Compl. temp	Compl. spaz
BFS	sì*	sì*	$O(b^d)$	$O(b^d)$
DFS	no	no	$O(b^m)$	$O(bm)$
UCS	sì*, [†]	sì	$O(b^{\lceil C^*/\varepsilon \rceil})$	$O(b^{\lceil C^*/\varepsilon \rceil})$
DLS	sì*, se $\ell \geq d$	sì*, se $\ell \geq d$	$O(b^\ell)$	$O(b\ell)$
IDS	sì*	sì*	$O(b^d)$	$O(bd)$

* completo se il branching factor b è un numero finito;

[†] completo se lo step cost è sempre maggiore di un certo valore soglia $\varepsilon > 0$;

★ ottimale se lo step cost è costante per ogni step.

Esempio

Vediamo una comparazione numerica tra BFS e IDS. Supponiamo un albero con $b = 3$ e $d = 3$ e calcoliamo la complessità spaziale effettiva:

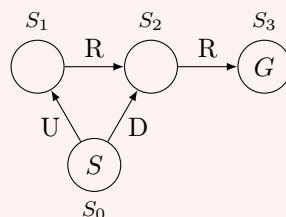
BFS ogni nodo può produrre al massimo b nodi figli, quindi ad ogni livello si produrranno b nodi per ogni nodo; si avranno quindi al livello d un numero di nodi pari a $b + b^2 + \dots + b^d$, che nel nostro caso sarà $3 + 3^2 + 3^3 = 39$;

IDS anche qui ogni nodo produce al massimo b nodi figli, ma in questo caso la produzione è ripetuta per ogni livello di profondità ℓ prima di arrivare a d , quindi il primo livello sarà generato d volte, il secondo $d-1$ e così via; si avranno quindi al livello d un numero di nodi pari a $db + (d-1)b^2 + \dots + b^d$, che nel nostro caso sarà $3 \cdot 3 + 2 \cdot 3^2 + 1 \cdot 3^3 = 54$.

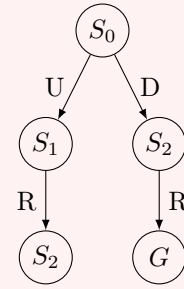
Come vediamo, IDS espande più volte i livelli e quindi risulta essere più oneroso dal punto di vista temporale, anche se appunto la complessità asintotica è la stessa.

Esempio

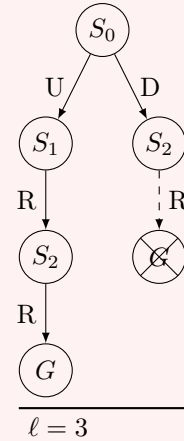
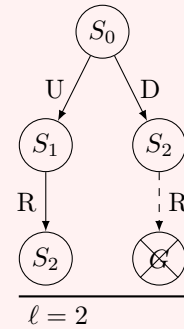
Applichiamo la strategie graph-search BFS e IDS nel seguente grafo:



BFS graph-search è ottimale? L'esplorazione avviene a partire da S_0 , il quale aggiunge alla frontiera S_1 ed S_2 in questo ordine *arbitrario*, per poi rimuoversi dalla frontiera e aggiungersi tra gli esplorati (questa operazione si ripete per ogni nodo che si espande). Supponendo di continuare con S_1 , lo si espande ma non si aggiunge S_2 alla frontiera poiché già presente. Ora si espande S_2 , il quale aggiunge S_3 , e finalmente l'algoritmo ha trovato il goal. In definitiva, BFS è ottimale nel caso graph-search, infatti il cammino trovato è il più corto.



IDS graph-search è ottimale? L'algoritmo itera la ricerca aumentando di volta in volta la profondità massima dell'espansione dei nodi; consideriamo per ora il caso $\ell = 2$, quindi ad algoritmo già partito. L'esplorazione avviene anche qui a partire da S_0 , il quale aggiunge S_1 ed S_2 nella frontiera in questo ordine *arbitrario*. Supponendo di continuare con S_1 , si procede in profondità espandendolo ma non aggiungendo S_2 alla frontiera. Dato che ora abbiamo raggiunto il livello 2 non possiamo espandere S_2 , ma esso viene comunque aggiunto ai nodi esplorati, e qui si ha il motivo del non raggiungimento del goal a questo livello, anche se G è effettivamente presente al livello 2. Infatti, proseguendo nella frontiera, si deve espandere ora S_2 , che però è già nella lista dei nodi esplorati, e quindi l'algoritmo termina per questo livello, non trovando la soluzione. Se fosse un caso di tree-search invece, non ci sarebbe la lista di nodi esplorati e quindi S_2 nel branch destro verrebbe espanso e si troverebbe il nodo per G . Ovviamente per $\ell = 3$ il goal viene raggiunto, dato che dal primo branch ora possiamo espandere S_2 e trovare appunto G , ma non viene raggiunto passando per il branch più corto (quello di destra) semplicemente perché, come detto prima, l'ordine di inserimento dei nodi espansi in frontiera è arbitrario e quindi non c'è garanzia di avere sempre il cammino più corto. In definitiva, **IDS non è ottimale nel caso graph-search.**



2.6 Strategie informed

Questo tipo di strategie vengono fornite di informazioni riguardo la relazione tra lo stato corrente dell'ambiente e il goal, ad esempio la distanza in linea d'aria tra la posizione attuale e quella desiderata: queste informazioni vengono dette **euristiche** e sono rappresentate come funzioni $h(n)$.

Una nuova famiglia di strategie è la **Best-First search**, che a differenza delle precedenti usa una **evaluation function** $h(n)$ su ogni nodo n per stimarne la *desiderabilità* e poi espande il nodo più desiderabile; la frontiera viene quindi implementata con una coda ordinata per desiderabilità decrescente. Le due strategie successive sono casi speciali di questo, con delle euristiche specifiche.

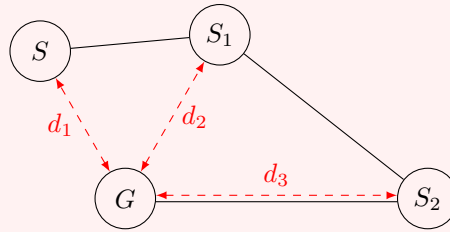
2.6.1 Greedy Best-First search

L'algoritmo usa un'euristica $h(n)$ su ogni nodo successore per stimare il costo per raggiungere il goal a partire da quel nodo e poi espande quello che *sembra* essere il meno costoso (è una UCS ordinata per euristica crescente).

Esempio

In una mappa stradale potremmo usare l'euristica *straight-line distance* $h_{SLD}(n)$ per misurare i chilometri in linea d'aria da una certa città a quella di arrivo, per poi scegliere tra le città vicine alla attuale quella che dista meno. Il problema di questa strategia è la non completezza, infatti se lo stato attuale si trova in un punto vicino al goal ma all'interno di un "vicolo cieco", il meglio che potrà fare sarà tornare indietro,

per poi inevitabilmente tornare al nodo precedente, poiché più vicino, e rimanere intrappolato in un loop; per evitare questo problema basta marcare i percorsi già esplorati.



Notiamo che a partire da S , l'algoritmo sceglierà S_1 , per poi scegliere di nuovo S , poiché ha distanza in linea d'aria minore rispetto a S_2 , per poi tornare in S_1 , e avanti così.

2.6.2 A^* search

Il costo è nella forma $f(n) = g(n) + h(n)$, dove

- $g(n)$ è il costo cumulativo per raggiungere il nodo n ;
- $h(n)$ è il costo stimato (euristica) per raggiungere il goal da n ;
- $f(n)$ è il costo stimato del cammino da n al goal.

In pratica il costo totale questa volta considera il costo effettivo per arrivare dove sarò dopo e anche il costo per andare dove voglio arrivare. Affinché la stima sia coerente con l'ambiente, è necessario usare una *euristica ammissibile*, che ci garantisce l'ottimalità di A^* .

Euristica ammissibile

Una euristica $h(n)$ è detta *ammissibile* se $0 \leq h(n) \leq h^*(n)$, cioè se la stima non supera mai il costo reale $h^*(n)$ e se $h(G) = 0$ per ogni goal G .

Osservazione: Quando si deve decidere tra un insieme di possibili euristiche quella che meglio si può adattare al problema si procede *togliendo vincoli*, cioè **rilassando** (semplificando) il problema, e poi si prende l'euristica che è sempre maggiore (per ogni nodo) di tutte le altre, perché si avrà la garanzia di non sottostimare mai il costo reale e quindi l'errore relativo rispetto al costo reale sarà minimo (approssima meglio delle altre).

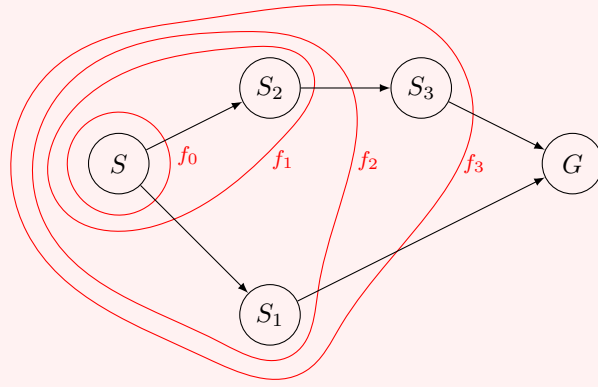
Esempio

Per il gioco del 8, tra le euristiche EUCLIDIANDISTANCE (distanza in linea d'aria tra un numero e la sua casella) e MANHATTANDISTANCE (numero di mosse ideali da effettuare per portare un numero alla sua casella, senza considerare la presenza degli altri numeri) è consigliato scegliere la seconda, in quanto il costo di spostamento libero è sempre maggiore o uguale al costo in linea d'aria e quindi l'approssimazione sarà migliore.

Possiamo vedere l'espansione di A^* come l'allargamento di un bordo f_i a partire dal nodo di partenza. Tale bordo, nel caso in cui l'euristica fosse una *consistente*, rappresenta la ricerca verso il prossimo stato ottimo e raggruppa al suo interno tutti i nodi con $f(n) \leq f_i$. Prima vengono espansi tutti i nodi entro f_i e poi si passa al prossimo bordo f_{i+1} . Dato che il costo complessivo $f(n)$ cresce, ovviamente si avrà $f_i < f_{i+1}$.

Esempio

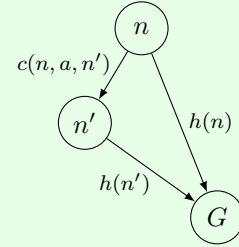
Usiamo nuovamente il caso del viaggio in auto. A^* prima espanderà tutti i nodi all'interno del bordo f_0 , poi di f_1 e così via, fino ad arrivare a G . Così facendo, il costo complessivo sarà sempre mantenuto al minimo.



Euristica consistente

Una euristica $h(n)$ è detta **consistente** se vale la condizione $h(n) \leq c(n, a, n') + h(n')$, dove $c(n, a, n')$ è il costo per andare da n a n' compiendo l'azione a . Questo significa che mano a mano che mi avvicino al goal l'euristica deve decrescere ed è in pratica la disuguaglianza dei triangoli, dove l'ipotenusa $h(n)$ è sempre minore della somma dei due cateti $c(n, a, n')$ e $h(n')$. Quando una euristica è consistente si ha che

$$\begin{aligned}
 f(n') &= g(n') + h(n') && \text{per definizione} \\
 &= g(n) + c(n, a, n') + h(n') && \text{scomposizione di } g(n') \\
 &\geq g(n) + h(n) && \text{dato che } h \text{ è consistente} \\
 &= f(n) && \text{per definizione}
 \end{aligned}$$



e quindi A^* espanderà sempre i nodi in ordine di costo non decrescente.

Osservazione: La consistenza implica l'ammissibilità, ma non il contrario:

$$\text{CONSISTENCY} \not\Rightarrow \text{ADMISSIBILITY}$$

Dimostrazione. Dimostriamo che $\text{CONSISTENCY} \not\Rightarrow \text{ADMISSIBILITY}$. In un grafo con m nodi indicizzati tramite i (oppure j), impostiamo la funzione di costo reale $c(i, j) = j - i$ (dove ogni arco ha costo 1) e una euristica $h(i) = m - 2\lceil \frac{i}{2} \rceil$.

Ammissibilità per la condizione $0 \leq h(i) \leq h^*(i)$ si ha $m - 2\lceil \frac{i}{2} \rceil \leq m - i$ e quindi $\lceil \frac{i}{2} \rceil \geq \frac{i}{2}$, che è vero per ogni i ;

Consistenza per la condizione $h(i) \leq c(i, i+1) + h(i+1)$ si ha $m - 2\lceil \frac{i}{2} \rceil \leq 1 + m - 2\lceil \frac{i+1}{2} \rceil$ e quindi $\lceil \frac{i}{2} \rceil \geq \frac{1}{2} + \lceil \frac{i}{2} + \frac{1}{2} \rceil$, che è falso per ogni i .

Così facendo, abbiamo dimostrato che utilizzando queste particolari funzioni A^* non riuscirà a trovare il cammino ottimo, poiché utilizza una metrica *possibile* ma non *concreta*. \square

Per quanto riguarda la complessità temporale, il tempo di esecuzione di A^* è

$$T_{A^*} = O((\text{errore relativo di } h(n) \text{ rispetto a } h^*(n) \times |\text{soluzione}|)^c)$$

per una certa costante c . Questo significa che il tempo di esecuzione è migliore quanto più è accurata l'euristica usata, un altro riscontro di come la sua scelta sia effettivamente una questione importante nell'approccio alla soluzione del problema.

A^* inoltre è detto **ottimamente efficiente**, ovvero tutti i nodi che vengono espansi per una certa euristica *dovevano essere espansi*, e non esiste un'altra ricerca che con la stessa euristica riesce a garantire di trovare sempre il cammino ottimo espandendo meno nodi. In particolare, prima di espandere nodi entro un certo bordo f_{i+1} , A^* espande prima tutti i nodi al bordo f_i .

Teorema

A^* trova sempre il cammino ottimo.

Dimostrazione. Dato un problema con due goal, uno ottimo G_1 ed uno sub-ottimo G_2 (ovvero il cui costo è maggiore rispetto a G_1), supponiamo che G_2 sia in frontiera e che n sia un nodo in frontiera e sul cammino più piccolo verso G_1 .

$f(G_2) = g(G_2) + h(G_2)$	per definizione
$= g(G_2)$	dato che $h(G_2) = 0$ poiché G_2 è un nodo goal
$> g(G_1)$	dato che G_2 è sub-ottimo
$= g(n) + h^*(n)$	scomposizione di $g(G_1)$
$\geq g(n) + h(n)$	dato che h è ammissibile
$= f(n)$	per definizione

Quindi, dato che $\forall n, f(G_2) > f(n)$, A^* non sceglierà mai G_2 per l'espansione, anche se è un nodo goal. \square

Osservazione: per essere ottimale, A^* graph-search ha bisogno che l'euristica sia consistente, mentre per A^* tree-search basta che sia ammissibile.

2.7 Problema - Lupo, Pecora e Cavolo: ricerca

Ora che abbiamo visto alcune strategie di risoluzione dei problemi, andiamo ad applicarle sul modello che abbiamo precedentemente costruito. In particolare, usiamo una strategia tree-search BFS sullo spazio degli stati (W, S, C, M) . Ricordiamo che lo stato iniziale è $(0, 0, 0, 0)$, il goal è $(1, 1, 1, 1)$ e le azioni sono:

$CARRYWOLF (W, S, C, M) \rightarrow (\overline{W}, S, C, \overline{M})$ se $S \neq C, W = M$;

$CARRYSHEEP (W, S, C, M) \rightarrow (W, \overline{S}, C, \overline{M})$ se $S = M$;

$CARRYCABBAGE (W, S, C, M) \rightarrow (W, S, \overline{C}, \overline{M})$ se $W \neq S, C = M$;

$CARRYNOTHING (W, S, C, M) \rightarrow (W, S, C, \overline{M})$ se $W \neq S, S \neq C$;

Espandiamo l'albero e troviamo la soluzione: inizialmente possiamo applicare solo CARRYSHEEP e poi tornare indietro con CARRYNOTHING; ora sono a un bivio, infatti posso applicare sia CARRYWOLF che CARRYCABBAGE, e la scelta ricade esclusivamente sul modo in cui vengono cercate le azioni applicabili allo stato: se la ricerca è lineare ad esempio, dall'elenco sopra eseguiremo prima CARRYWOLF. In questo caso, entrambe le scelte sono ammesse, infatti il problema ha due soluzioni

- {CSHEEP, CNOTHING, CCABBAGE, CSHEEP, CWOLF, CNOTHING, CSHEEP};
- {CSHEEP, CNOTHING, CWOLF, CSHEEP, CCABBAGE, CNOTHING, CSHEEP}.

ALTRO PROBLEMA DI MAP NAVIGATION SUL QUADERNO

3 | Problemi di ottimizzazione

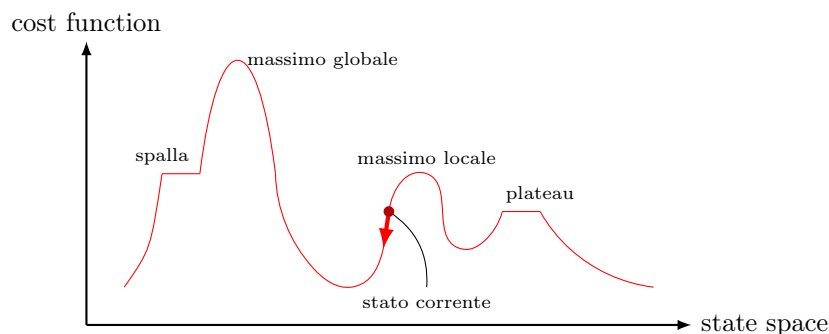
Per alcuni problemi non è importante il come arrivare allo stato desiderato, ma il capire quale esso sia, un esempio sono la schedulazione, il design di circuiti integrati o la creazione di un itinerario in una mappa.

3.1 Strategie locali

Mediante queste strategie si ignora completamente il percorso per arrivare alla soluzione e invece *si cerca la soluzione* in sé, utilizzando un unico **stato corrente** e cercando di muoversi nei suoi dintorni per trovare la soluzione migliore. Questi algoritmi saranno quindi costanti nella complessità spaziale (e anche in quella temporale, vedremo perché) e potranno essere usati sia per ricerca online sia per offline, essendo sempre ottimi a meno di 1%. Lo spostamento nei dintorni avverrà in base ad una **funzione obiettivo**, cioè un'euristica che associa ad ogni stato il suo grado di *desiderabilità*, oppure alla già nota **funzione di costo**.

3.1.1 Hill-Climbing (o gradient descent)

Per comprendere meglio il concetto di *dintorni* tracciamo un grafico della funzione euristica nello spazio degli stati, così da descrivere lo **state-space landscape** del problema: se l'euristica è una **funzione di costo**, cercheremo la valle più profonda (**minimo globale**) mentre se l'euristica è una **funzione obiettivo**, cercheremo la montagna più alta (**massimo globale**); in entrambi i casi, cercheremo lo stato migliore tra i vicini del corrente, muovendoci in salita o in discesa sul landscape e cambiando di volta in volta lo stato corrente.



Algorithm 8 Hill-Climbing

```
1: function HILLCLIMBING(problem)
2:   current = MAKENODE(problem.InitialState)
3:   loop
4:     neighborhood = EXPAND(current,problem)
5:     neighbor = MAX(neighborhood, elem → elem.Value)
6:     if neighbor.Value ≤ current.Value then return current.State
7:     current = neighbor
```

Visto che l'algoritmo ha tempo costante, possiamo ripeterlo molte volte partendo da punti casuali del landscape (**random restart**), e selezionare il miglior risultato. Quando abbiamo dei vicini sempre con lo stesso valore (punti *spalla* oppure *plateau*) possiamo eseguire dei piccoli spostamenti casuali nell'intorno dello stato corrente (**random sideways moves**) per evitare di rimanere incastrati nei punti di spalla e trovare il nuovo massimo/minimo, rimanendo però in un loop nel caso incappassimo in un plateau (perché l'algoritmo ad ogni spostamento troverà sempre lo stesso valore massimo/minimo locale e cercando di scappare da esso troverebbe solo le discese/salite laterali, che lo farebbero tornare nuovamente a spostarsi nell'intorno).

3.1.2 Simulated Annealing

Si ispira alla meccanica statistica: facciamo delle sempre meno frequenti scelte random per cercare di spostarci dal massimo/minimo locale. Chiamiamo **temperatura** l'intensità delle mosse casuali rispetto a quelle sensate (viste nell'algoritmo Hill-Climbing) e ci spostiamo in base all'energia sprigionata dalla differenza.

In pratica più è alta la temperatura più si saltella in giro per il landscape in preda al panico, poiché l'energia sprigionata è tanta, ma quando la temperatura diminuisce ci si stabilizza verso un certo stato con energia minima, andando a rinfrescare i piedi ustionati.

Algorithm 9 Simulated Annealing algorithm

```

1: function SIMULATEDANNEALING(problem,schedule)
2:   current = MAKENODE(problem.InitialState)
3:   for time  $\leftarrow 1$  to  $\infty$  do
4:     // assegna una temperatura in base al tempo, diminuendola pian piano
5:      $T \leftarrow \text{schedule}(\text{time})$ 
6:     if  $T = 0$  then return current
7:     neighborhood  $\leftarrow \text{EXPAND}(\text{current}, \text{problem})$ 
8:     next  $\leftarrow \text{RANDOM}(\text{neighborhood})$ 
9:     // se l'energia è positiva si salta
10:    // se l'energia è negativa c'è qualche possibilità di saltare
11:     $\Delta E \leftarrow \text{next.Value} - \text{current.Value}$ 
12:    if  $\Delta E > 0$  then return current  $\leftarrow$  next
13:    else current  $\leftarrow$  next, con probabilità  $e^{\Delta E/T}$ 

```

Concentriamoci sull'ottimalità ora: a temperatura fissa T , la probabilità di trovarsi in un certo stato x è la distribuzione di Boltzman

$$p(x) = \alpha e^{\frac{E(x)}{kT}}$$

mentre invece se T decresce *sufficientemente piano* si può dimostrare che si raggiunge sempre il miglior stato x^* , infatti per T *piccola* la probabilità di trovarsi in x^* rispetto a quella di trovarsi in un generico x è

$$e^{\frac{E(x^*)}{kT}} / e^{\frac{E(x)}{kT}} = e^{\frac{E(x^*) - E(x)}{kT}} \gg 1.$$

L'unico inghippo è che non possiamo garantire l'ottimalità di questo algoritmo perché la velocità di decrescita di T dipende dallo *schedule* utilizzato dal problema e per lo stesso motivo non abbiamo garanzie nemmeno sul fatto che T sia piccola. Scelto però un buon *schedule*, il simulated annealing è uno dei migliori algoritmi da utilizzare nei casi reali proprio perché è semplice da realizzare ed è molto efficiente, dando praticamente sempre buoni risultati.

3.1.3 Local Beam Search

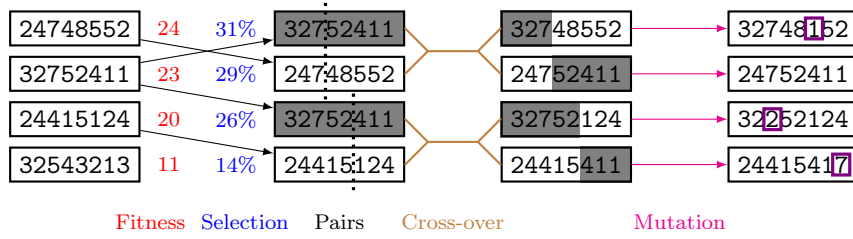
Si basa sul Hill-Climbing, ma anziché avere un solo stato corrente ne ha k : per ognuno dei k stati (detti *particelle*) si estraggono i successori, come di consueto, ma questa volta tra tutti quelli estratti ne vengono selezionati k tra i migliori, con i quali proseguire. In pratica la ricerca comincia su più fronti ma converge verso l'unico più promettente (attenzione, non è una ricerca *in parallelo* su più fronti, ma piuttosto una osservazione su più fronti, che successivamente si interrompe se non fruttuosa o che prosegue con più intensità se redditizia).

Spesso tutti i k stati si incastrano in un massimo/minimo locale: per evitare questo inghippo, la selezione dei k nuovi stati viene fatta in maniera leggermente casuale, dando la possibilità anche agli stati che non sono tra i primi k in classifica di essere scelti come successori, e rendendo quindi possibile l'evasione dal punto di massimo/minimo.

Possiamo vedere queste piccole casualità come lo *spirito di sopravvivenza* dei nuovi stati, il che rende lampante l'analogia con i processi di *selezione naturale*: in una comunità di k individui, solo i k più forti sopravvivono (per mantenere l'equilibrio nascite/decessi e utilizzare sempre lo stesso quantitativo di risorse), mentre alcuni individui, seppur più deboli, riescono ad ingegnarsi per rimanere in vita, e quindi si ritrovano tra i k della generazione successiva.

3.1.4 Algoritmi genetici

Nel caso della Local Beam Search abbiamo *individui forti* e *individui fortunati*, il che divide gli alberi di discendenti in due categorie di stati con proprietà entrambe interessanti: dato che alla fine solo uno di questi alberi sopravvivrà, vorremmo riuscire a miscelarli per ottenere sempre l'*individuo perfetto*, unendo forza e fortuna. Se prima era lampante la selezione naturale, ora raffiniamo questa analogia ai *processi genetici*, con i quali possiamo introdurre *accoppiamenti*, nei quali si scambiano porzioni di soluzioni e *mutazioni*, con le quali vengono casualmente modificati gli stati.



L'algoritmo è suddiviso in più step: alcuni stati iniziali arbitrari vengono etichettati usando una funzione di *fitness*, che indica quanto è *adatto* uno specifico stato, e poi *sopravvivono* in base a una certa funzione di probabilità legata al loro fitness, simulando una *selezione naturale*; gli individui vincenti si *accoppiano* tra loro, *incrociando* le proprie caratteristiche in maniera casuale, ed infine possono subire una *mutazione*, che altera una loro particolare caratteristica; il processo viene ripetuto per un numero arbitrario di *generazioni*, che convergono verso l'individuo perfetto.

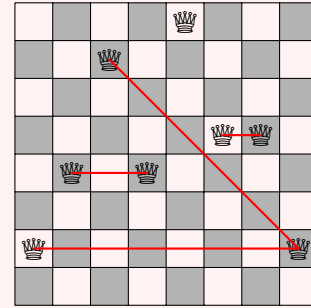
Esempio

Nella figura sopra viene trattato il problema delle 8 regine, le quali devono essere posizionate sulla scacchiera in modo da non mangiarsi tra loro. Uno stato è una lista di 8 valori numerici, uno per ogni colonna, che indicano l'indice della riga in cui si trova la regina per quella colonna, dal basso verso l'alto. Definiamo la *funzione di fitness* in modo che sia più alta quando i conflitti tra regine sono più bassi, quindi uso il numero di conflitti totali c_M come limite superiore; per calcolare questo valore possiamo pensare alle regine come un grafo a 8 nodi completamente connesso, ovvero con tutte le combinazioni di archi non diretti possibili

$$c_M = C_{8,2} = \frac{8!}{(8-2)! \cdot 2!} = \frac{8 \cdot 7}{2} = 28$$

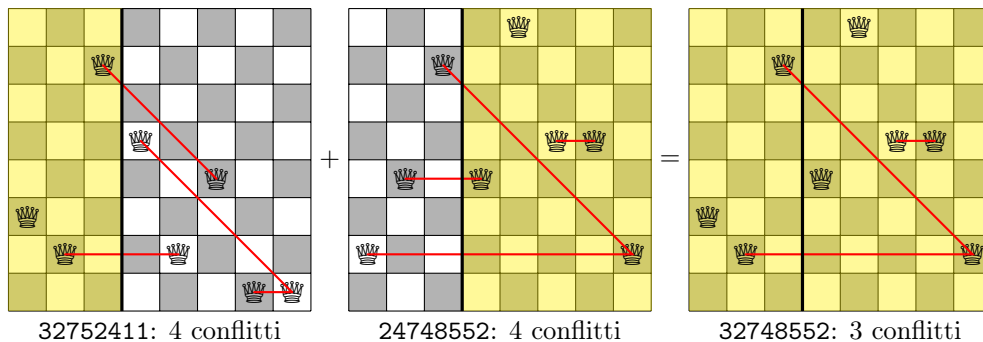
e quindi la funzione di fitness per un certo stato i ritornerà $F_i = c_M - c_i$. Nel nostro caso, per lo stato 24748552 contiamo 4 conflitti, quindi avremo un fitness $F_{24748552} = 28 - 4 = 24$.

La *funzione di selezione* viene costruita per essere la percentuale della fitness di un certo individuo i rispetto al totale di tutte le fitness, in pratica è $P_i = \frac{F_i}{\sum_i F_i}$.



Come vediamo sono necessari molti parametri (come scelgo la funzione di fitness? e quella di selection? dove taglio le coppie? e quanti tagli faccio? con quale frequenza ammetto mutazioni? e con quanta intensità?), ed è proprio per questo che gli algoritmi genetici risultano essere particolarmente delicati da impostare e sono specifici del caso in cui vengono utilizzati; d'altra parte però, quando ben impostati, permettono di ottenere ottimi risultati che approssimano in modo eccellente l'ideale *evoluzione naturale* dello stato.

Gli algoritmi genetici sono particolarmente indicati quando si può suddividere lo stato in *porzioni* da utilizzare per il cross-over, il quale mescola appunto le *caratteristiche* utili: in pratica nel caso delle 8 regine, la scacchiera si può tagliare lungo l'altezza e i vari conflitti da un lato all'altro del taglio si possono eliminare, preservando quelli all'interno delle due metà: in questo modo si ha una possibilità di unire aspetti positivi di due stati, ottenendo uno stato migliore dei due precedenti da qui proviene, oppure di ottenerne uno peggiore, che comunque non sopravviverà alla prossima selezione.



3.1.5 Ricerca locale in spazi continui

Gli spazi degli stati visti finora sono sempre stati discreti, ma se invece abbiamo degli spazi continui le cose si fanno più complicate: prendiamo ad esempio il problema di posizionare tre aeroporti in modo tale che le

distanze dell'aeroporto dalle città più vicine siano le minori possibili (cioè posizionare gli aeroporti in modo che la raggiungibilità sia massima). Lo stato è un vettore $x = (x_1, y_1, x_2, y_2, x_3, y_3)$ mentre la funzione obiettivo è

$$f(x_1, y_1, x_2, y_2, x_3, y_3) = \sum_{i=1}^3 \sum_{c \in C_i} (x_i - x_c)^2 + (y_i - y_c)^2$$

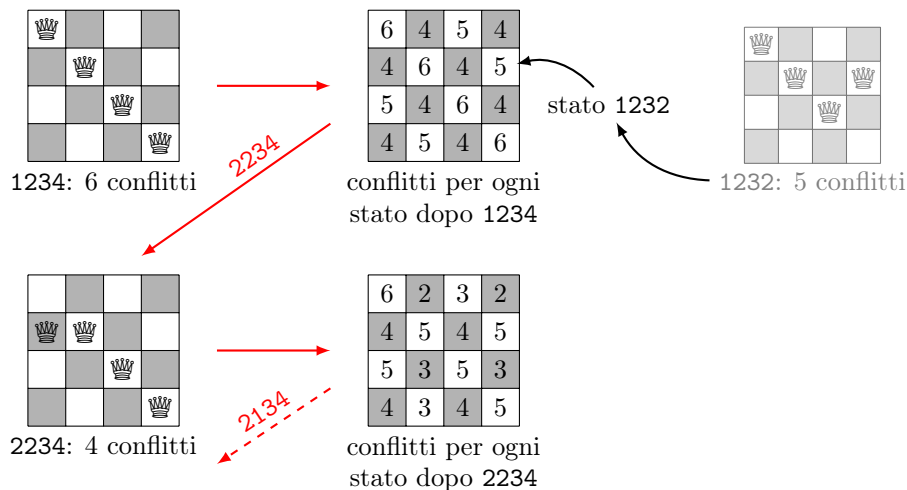
dove C_i sono le città più vicine all' i -esimo aeroporto. Usando i metodi visti finora, lo spazio degli stati esploderebbe, dato che non possiamo rappresentare il dominio reale in un calcolatore: discretizziamo il problema in modo che ci si possa ricondurre ad una soluzione simile al Hill-Climbing, dove ogni cambio di coordinata ha passo $\pm\delta$. Dall'analisi matematica, sappiamo che per minimizzare le distanze in f possiamo cercare di annullare il gradiente, quindi vogliamo

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial y_1}, \frac{\partial f}{\partial x_2}, \frac{\partial f}{\partial y_2}, \frac{\partial f}{\partial x_3}, \frac{\partial f}{\partial y_3} \right) = 0$$

ma in questo modo purtroppo riusciamo a trovare la soluzione solo *localmente* (e non globalmente), in quanto le città vicine cambiano ogni volta che spostiamo gli aeroporti, cioè ogni volta che aggiorniamo lo stato. Lo stato viene aggiornato usando $x \leftarrow x + \alpha \nabla f(x)$, dove α ha valore migliore quando è l'hessiana di f : secondo Newton-Raphson, sarebbe meglio aggiornare x usando $x \leftarrow x - H_f^{-1}(x) \nabla f(x)$ dove $H_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}$.

3.2 Problema - 4 Regine

Vogliamo risolvere una versione semplificata del problema visto sopra, nel quale ora 4 regine devono essere posizionate su una scacchiera 4×4 in modo che non ci siano conflitti tra loro. Usando un approccio Hill-Climbing, partiamo dallo stato 1234 (le righe sono ordinate dall'alto verso il basso) e scriviamo per ogni cella di ogni colonna il numero di conflitti che si verificano spostando la regina di quella colonna in quella cella



e a questo punto scegliamo lo stato che ci da meno conflitti, in questo caso uno stato arbitrario tra quelli da 4 conflitti (ad esempio 2234), poi si ripete il processo fino a trovare uno stato con 0 conflitti.

4 | Constraint Processing

Se nel problema sono presenti dei vincoli evidenti nel dominio (ad esempio, nel problema della navigazione non ci si può muovere in una casella segnata come muro), possiamo sfruttarli per evitare di sprecare tempo visitando tanti stati non validi a causa di tali vincoli. Queste tecniche si adattano particolarmente bene ai problemi combinatori, nei quali esiste sempre una soluzione ma sarebbe impossibile trovare la migliore in tempo utile (bisogna controllare tutti i possibili stati, che sono un numero esponenziale, alle volte anche fattoriale).

Esempio

Per il problema delle 4 regine possiamo rappresentare i vincoli usando 4 coppie di variabili x_i, y_i per identificare la posizione di ogni regina sulla scacchiera e formalizzare i vincoli in questo modo:

- vincolo righe diverse: $x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_2 \neq x_3, x_2 \neq x_4, x_3 \neq x_4$;
- vincolo colonne diverse: $y_1 \neq y_2, y_1 \neq y_3, y_1 \neq y_4, y_2 \neq y_3, y_2 \neq y_4, y_3 \neq y_4$;
- vincolo diagonali diverse: $|x_1 - x_2| \neq |y_1 - y_2|, |x_1 - x_3| \neq |y_1 - y_3|, \dots$

ma possiamo fare ancora meglio: se anziché avere una variabile per la riga e una per la colonna, dato che sappiamo che ci sarà sempre una e una sola regina per ogni colonna, possiamo avere una sola variabile per l'altezza a partire dall'alto (stiamo praticamente cablando un vincolo nel modello dello stato, come abbiamo fatto in precedenza), quindi usiamo 4 variabili r_i per identificare la riga sulla quale la i -esima regina viene posizionata e i vincoli diventano più compatti:

- vincolo righe diverse: $r_1 \neq r_2, r_1 \neq r_3, r_1 \neq r_4, r_2 \neq r_3, r_2 \neq r_4, r_3 \neq r_4$;
- vincolo colonne diverse: cablato nel modello dello stato, posso ignorarlo;
- vincolo diagonali diverse: $|r_1 - r_2| \neq 1, |r_1 - r_3| \neq 2, \dots, |r_i - r_j| \neq |i - j|$.

4.1 Reti a vincoli

Per formalizzare questa tecnica, definiamo il concetto di **rete a vincoli**, un modello grafico che usa un grafo per rappresentare i vincoli tra gli stati. Una rete a vincoli è una tripla $\mathcal{R} = (X, D, C)$, dove

- X un insieme di n variabili x_i ;
- D un insieme di n domini $D_i = \{v_1 \dots v_{k_i}\}$ dei k_i valori assegnabili alla variabile x_i ;
- C un insieme di vincoli $C_j = (S_j, R_j)$ con $S_j \subseteq X$ e $R_j \subseteq S_j \times S_j$ dove
 - ★ S_j sottoinsieme di variabili (**scope**) legate dal vincolo;
 - ★ R_j insieme di tuple accettate (**relation**) come valori per S_j ;

Con questo formalismo possiamo costruire due grafi:

grafo primale del problema, dove ogni vertice rappresenta una variabile e ogni arco rappresenta un vincolo; è una rappresentazione abbastanza superficiale della relazione tra le variabili, ma evidenzia bene le variabili con il cui assegnamento risulta critico;

grafo duale del problema, dove ogni vertice rappresenta uno scope di un vincolo e ogni arco rappresenta le variabili in comune tra quegli scope; risulta particolarmente utile poiché rappresenta esplicitamente tutte le catene di vincoli nel problema.

Utilizzando algoritmi di propagazione dei vincoli e ricerca sui grafi, riusciremo a trovare *una soluzione* oppure *tutte le soluzioni* (ricerca), *la migliore soluzione* (ottimizzazione), *la conta delle soluzioni* o compiere un *controllo di consistenza*.

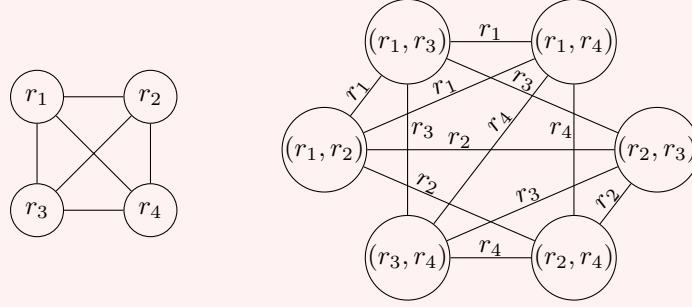
Esempio

Sempre per il problema delle 4 regine, possiamo ora utilizzare il formalismo delle reti a vincoli e descrivere il problema come

- $X = \{r_1, r_2, r_3, r_4\}$;

- $D = \{D_1, D_2, D_3, D_4\}$ dove $\forall i, D_i = \{1, 2, 3, 4\}$
- $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ dove
 - ★ $C_1 = (\{r_1, r_2\}, \{(1, 3), (1, 4), (2, 4), (3, 1), (4, 1), (4, 2)\})$;
 - ★ $C_2 = (\{r_1, r_3\}, \{(1, 2), (1, 4), (2, 1), (2, 3), (3, 2), (3, 4), (4, 1), (4, 3)\})$;
 - ★ $C_3 = (\{r_1, r_4\}, \{(1, 2), (1, 3), (2, 1), (2, 3), (2, 4), (3, 1), (3, 2), (3, 4), (4, 2), (4, 3)\})$;
 - ★ $C_4 = (\{r_2, r_3\}, \{(1, 3), (1, 4), (2, 4), (3, 1), (4, 1), (4, 2)\})$;
 - ★ $C_5 = (\{r_2, r_4\}, \{(1, 2), (1, 4), (2, 1), (2, 3), (3, 2), (3, 4), (4, 1), (4, 3)\})$;
 - ★ $C_6 = (\{r_3, r_4\}, \{(1, 3), (1, 4), (2, 4), (3, 1), (4, 1), (4, 2)\})$.

Possiamo inoltre costruire il grafo primale e duale per il problema



Una rete a vincoli permette un approccio parziale al problema: possiamo considerare un sottoinsieme di variabili e solamente i vincoli presenti tra loro per trovare delle **soluzioni parziali**, ovvero soluzioni *rilassate* libere dai vincoli delle variabili escluse dal sottoinsieme. Tali soluzioni parziali possono essere **consistenti**, cioè valide considerando il sottoinsieme di vincoli, ma potrebbero non condurre alla soluzione globale, poiché potrebbero essere semplicemente consistenti localmente (proprio perché non considerano tutti gli altri vincoli).

4.1.1 Rappresentazione dei vincoli

Per i vincoli possiamo utilizzare più formalismi, più o meno compatti:

tabelle (o anche **tuple**) vengono elencate tutte le tuple accettate dal vincolo;

espressioni aritmetiche esprimono le caratteristiche delle tuple con formule compatte, senza elencarle esplicitamente come nelle tabelle;

formule proposizionali esprimono la relazione tra le variabili con la logica booleana.

Esempio

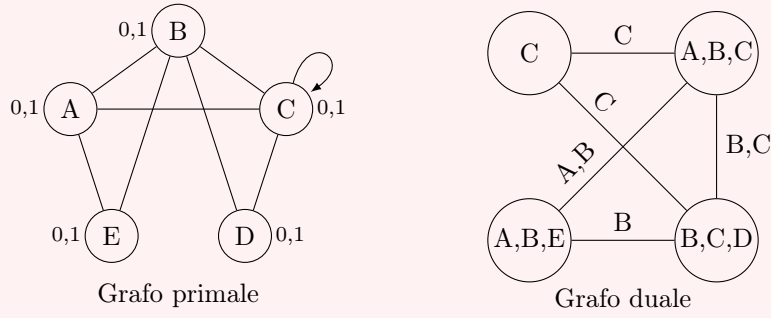
L'assegnazione a soluzione del problema SAT sulla formula CNF di clausole C_i

$$\phi(A, B, C, D, E) = \underbrace{(\neg C)}_{C_1} \wedge \underbrace{(A \vee B \vee C)}_{C_2} \wedge \underbrace{(\neg A \vee B \vee E)}_{C_3} \wedge \underbrace{(\neg B \vee C \vee D)}_{C_4}$$

può essere trovata utilizzando constraint processing, dove le clausole C_i saranno i nostri vincoli C_i , poiché devono tutte avere valore 1. Creiamo la rete a vincoli:

- $X = \{A, B, C, D, E\}$;
- $D = \{D_A, D_B, D_C, D_D, D_E\}$ dove $\forall \alpha, D_\alpha = \{0, 1\}$;
- $C = \{C_1, C_2, C_3, C_4\}$ vincoli delle clausole, con rappresentazione a tuple
 - ★ $C_1 = (\{C\}, (0))$;
 - ★ $C_2 = (\{A, B, C\}, \text{ALL} \setminus (0, 0, 0))$;
 - ★ $C_3 = (\{A, B, E\}, \text{ALL} \setminus (1, 0, 0))$;
 - ★ $C_4 = (\{B, C, D\}, \text{ALL} \setminus (1, 0, 0))$;

e rappresentiamo i grafi primale e duale:



4.1.2 Tecniche di soluzione

Una volta impostata la rete a vincoli, dobbiamo trovare lo stato che risolve il problema, e per farlo possiamo usare due tecniche:

Inferenza vengono *propagati* i vincoli ed eliminati i valori dai domini delle variabili che non soddisfano i vincoli. In questo modo otteniamo una rete più *stretta*, poiché con meno tuple accettabili, e gli assegnamenti parziali non consistenti possono essere scartati senza dover esplorare il loro sottoalbero di soluzioni, poiché già sappiamo che alla fine non rispetteranno i vincoli;

Ricerca si comincia come al solito da uno stato casuale e cambiando uno a uno i valori delle variabili ci si sposta verso gli stati che violano sempre meno vincoli.

Per quanto detto, per trovare la soluzione sarebbe quindi logico inferire il più possibile e poi assegnare un valore ad una variabile arbitraria e seguire la catena di vincoli che da essa partono: purtroppo questo non è fattibile, infatti il costo spaziale e temporale dell'inferenza è esponenziale, quindi è necessario fermarsi ad un certo punto ed eseguire la ricerca classica. Dato che anche gli algoritmi di ricerca hanno costo esponenziale, solitamente si usa una combinazione dei due metodi, prima propagando i vincoli tramite inferenza per eliminare la maggior parte delle tuple non accettabili e poi ricercando la soluzione tra le rimanenti.

4.2 Propagazione dei vincoli

Dato che le relazioni dei vincoli sono sottoinsiemi del prodotto cartesiano delle variabili nello scope, possiamo applicare tutte le operazioni standard degli insiemi sulle relazioni (quelle dell'algebra relazionale, **intersezione** $R_1 \cap R_2$, **unione** $R_1 \cup R_2$, **differenza** $R_1 - R_2$, **selezione** $\sigma_{\text{cond}}(R)$, **proiezione** $\pi_{\text{vars}}(R)$, **join** $R_1 \bowtie R_2$, ricordando che la proiezione elimina i duplicati).

Per inferire vincoli tra variabili x_i, x_j in relazioni diverse R_1, R_2 è sufficiente utilizzare $\pi_{x_i, x_j}(R_1 \bowtie R_2)$; il processo di inferenza va a propagare i vincoli esistenti aggiungendo alla rete **vincoli ridondanti** e generando quindi una **rete equivalente** alla precedente, cioè con stesse variabili e soluzioni; i vincoli ridondanti non sono però da considerare come inutili, infatti vengono utilizzati per rendere evidenti relazioni che prima erano implicite e semplificare la ricerca della soluzione, riducendo il numero di tuple possibili tra le quali ricercare.

La propagazione dei vincoli avviene considerando soluzioni parziali sempre più grandi e forzando le condizioni di **consistenza locale**

node consistency se ogni vincolo unario di ogni variabile x_i (è uno scope a un sola variabile) viene soddisfatto da tutti i valori nel suo dominio D_i (è una 1-consistency). In formule, $\forall v \in D_i, (v) \in R_{x_i}$ in ogni vincolo $C = (\{x_i\}, R_{x_i})$;

arc consistency se ogni vincolo binario di ogni scope a due variabili $\{x_i, x_j\}$ viene soddisfatto da tutte le tuple ottenute dai loro domini D_i, D_j (è una 2-consistency). In formule, $\forall u \in D_i, \exists v \in D_j. (u, v) \in R_{x_i, x_j}$;

path consistency se ogni vincolo ternario di ogni scope a tre variabili $\{x_i, x_j, x_k\}$ viene soddisfatto da tutte le triple ottenute dai loro domini D_i, D_j, D_k (è una 3-consistency);

i-consistency se ogni vincolo n -ario di ogni scope a n variabili viene soddisfatto da tutte le tuple ottenute dai loro domini.

In pratica per verificare la i -consistency basta verificare che una soluzione parziale con $i - 1$ variabili sia $i - 1$ -consistent e poi estenderla a una soluzione parziale con i variabili in modo che per ogni valore legale (cioè nel dominio) di ognuna delle $i - 1$ variabili si possa trovare un valore per ogni altra variabile a lei connessa; se ciò non è possibile, non si può dichiarare la i -consistency;

Diciamo che una rete è **globalmente consistente** se è i -consistente per $i = 1 \dots n$ dove n è il numero di variabili del problema: questo mi assicura che assegnando variabili in ordine arbitrario ma rispettando i vincoli riuscirò sempre a trovare una soluzione, senza dover tornare indietro (cioè *backtrack free*) e cambiare un valore precedentemente assegnato perché mi ha portato ad una inconsistenza logica.

4.3 Algoritmi di propagazione

Per rendere una rete i -consistente dobbiamo forzare le condizioni di i -consistency e per farlo si eliminano tutte le tuple che non le soddisfano:

- per la *node consistency* la formula $D'_i = D_i \setminus \{v \mid \exists C = (\{x_i\}, R_{x_i}) \wedge v \notin R_{x_i}\}$ esprime l'eliminazione, infatti D'_i contiene solo valori che rispettano il vincolo e la node consistency;
- per la *arc consistency* possiamo definire la procedura

Algorithm 10 REVISE($(x_i)x_j$)

Require: R_{x_i,x_j}, D_i, D_j

Ensure: D_i tale che x_i è arc consistent rispetto a x_j

```

1: for all  $u \in D_i$  do
2:   if  $\nexists v \in D_j \mid (u, v) \in R_{x_i,x_j}$  then  $D_i \leftarrow D_i \setminus \{u\}$ 
```

che esprime in maniera compatta la formula $D_i \leftarrow D_i \cap \pi_{x_i}(R_{x_i,x_j} \bowtie D_j)$, cioè la definizione di inferenza, ed elimina i valori che non sono accettabili da un lato di un vincolo: per verificare arc consistency usiamo

Algorithm 11 AC-1

Require: $\mathcal{R} = \{X, D, C\}$

Ensure: \mathcal{R}' rete arc consistent più completa per \mathcal{R}

```

1: repeat
2:   for all  $x_i, x_j$  che partecipano a un vincolo do
3:     REVISE( $(x_i)x_j$ )
4:     REVISE( $(x_j)x_i$ )
5: until nessuna modifica nei domini
```

la cui terminazione è sempre garantita, poiché ① se non si modificano i domini termina, ② se rimuoviamo un valore almeno un dominio viene modificato e ③ se un dominio è vuoto termina senza soluzione.

Al crescere del grado il concetto di consistenza diventa sempre meno intuitivo, quindi ci fermiamo qui; ci concentreremo principalmente sulla arc consistency, andando a studiare diverse versioni dell'algoritmo di propagazione.

Esempio

Dato il seguente problema

$$X = \{x, y, z\} \text{ con } D_x = D_y = D_z = \{1, 2, 3\}$$

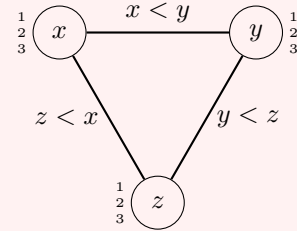
$$C = \{x < y, y < z, z < x\}$$

utilizziamo l'algoritmo AC-1 per propagarne i vincoli, mostrando ogni chiamata a REVISE:

```

REVISE( $(x)y$ )  $\rightarrow D_x = \{1, 2\}$ 
REVISE( $(y)x$ )  $\rightarrow D_y = \{2, 3\}$ 
REVISE( $(y)z$ )  $\rightarrow D_y = \{2\}$ 
REVISE( $(z)y$ )  $\rightarrow D_z = \{3\}$ 
REVISE( $(z)x$ )  $\rightarrow D_z = \{\}$ , STOP
```

Notiamo che non viene eseguito REVISE($(x)z$) poiché abbiamo ottenuto un dominio vuoto e l'algoritmo termina senza soluzione.



Il costo di AC-1 è $O(|V||E||D|^3)$, dove $|V|$ è il numero di vertici, $|E|$ il numero di archi e $|D|$ il massimo numero di valori nei domini: il costo è tale perché per ogni ciclo facciamo $2|E||D|^2$ operazioni (poiché per ogni nodo in ogni coppia di variabili in ogni arco, cioè $2|E|$ volte, REVISE effettua un controllo incrociato dei

domini, con costo $|D|^2$) e nel caso peggiore viene eliminato un solo valore per ogni ciclo, quindi possiamo avere al massimo $|V||D|$ cicli.

La complessità di AC-1 è cubica nei domini ma possiamo fare di meglio: utilizziamo un nuovo algoritmo che usa una coda per gestire le catene di vincoli da ricontrollare quando un dominio viene cambiato

Algorithm 12 AC-3

Require: rete a vincoli $\mathcal{R} = \{X, D, C\}$

Ensure: \mathcal{R}' rete arc consistent più completa per \mathcal{R}

```

1:  $Q \leftarrow \{\}$ 
2: for all  $x_i, x_j$  che partecipano a un vincolo do
3:    $Q \leftarrow Q \cup \{(x_i, x_j), (x_j, x_i)\}$ 
4: while  $Q \neq \{\}$  do
5:    $(x_i, x_j) \leftarrow \text{POP}(Q)$ 
6:    $\text{REVISE}((x_i), x_j)$ 
7:   if  $D_i$  è stato modificato then
8:     // aggiungi le coppie in cui  $x_i$  aggiorna una variabile diversa da  $x_j$ 
9:      $Q \leftarrow Q \cup \{(x_k, x_i) \mid \forall x_k, k \neq i, k \neq j\}$ 

```

Il costo di AC-3 è $O(|E||D|^3)$, non viene rimosso il cubo ma almeno non abbiamo più l'influenza dei nodi: il costo è tale perché REVISE costa ancora $|D|^2$, ma ora nel caso peggiore viene usata al massimo $2|E||D|$ volte perché le coppie vengono rimesse in coda al massimo $|D|$ volte.

Questo non è ancora il miglior algoritmo (esiste AC-4, che ha costo quadratico), ma tutti sono comunque polinomiali, quindi AC-3 è già abbastanza efficiente e continueremo ad usarlo come algoritmo principale.

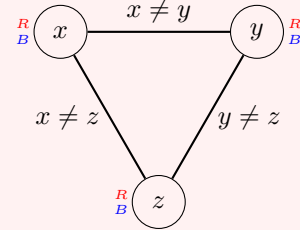
Esempio

Dato il problema di 2-colorazione di un grafo con 3 nodi, dove quindi

$$X = \{x, y, z\} \text{ con } D_x = D_y = D_z = \{R, B\}$$

$$C = \{x \neq y, y \neq z, x \neq z\}$$

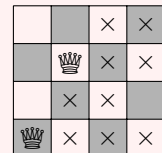
utilizziamo l'algoritmo AC-1 per propagare i vincoli e notiamo che i domini non cambiano, quindi la rete è già arc consistent (poiché per ogni assegnamento di una variabile tutte le altre ne hanno uno che non viola i vincoli), ma è immediato vedere che non esiste nessun assegnamento alle variabili che permetta di soddisfare tutti i vincoli (poiché sto cercando di colorare una clique di 3 nodi solamente con 2 colori), quindi il problema è globalmente inconsistente.



Osservazione: Se una rete è arc consistent *non necessariamente* è anche consistente: abbiamo visto che se forzando la arc consistency otteniamo un domino vuoto allora il problema è inconsistente e non ha soluzione, ma se forzando la arc consistency tutti i domini non sono vuoti non è detto che il problema abbia una soluzione, perché non sappiamo se sia globalmente consistente, infatti è necessario verificare ogni i -consistency per dirlo (lo avevamo detto anche in precedenza).

Esempio

Per il problema delle 4 regine notiamo che l'assegnamento parziale 13.. è localmente consistente (in particolare arc consistente), infatti non ci sono conflitti tra le due regine nella figura, ma impedisce la consistenza globale, infatti a causa di questi due assegnamenti non è possibile assegnare nessun valore per la terza regina.



Osservazione: In merito all'osservazione precedente, per alcuni problemi la arc consistency implica la consistenza globale: questi problemi vengono detti **casi trattabili**, poiché sono risolvibili in tempo polinomiale. Un problema è un *caso trattabile* se e solo se ha ① nessun dominio vuoto, ② solo vincoli binari, ③ grafo primale aciclico; per risolverlo polinomialmente basta forzare la arc consistency e a cominciare da una variabile arbitraria, dargli un valore nel suo dominio e seguire la catena di vincoli assegnando valori opportuni a tutti gli altri nodi, per poi costruire la soluzione con questi valori assegnati; come al solito, se durante la propagazione un dominio è diventato vuoto, il problema risulta inconsistente e quindi non ha soluzione.

Esempio

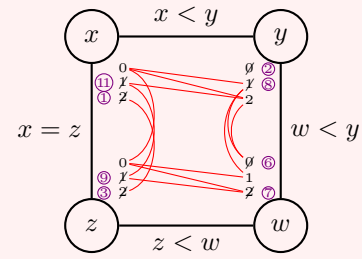
Dato il problema

$$X = \{x, y, z, w\} \text{ con } D_{x,y,z,w} = \{0, 1, 2\}$$

$$C = \{x < y, x = z, z < w, w < y\}$$

disegniamone il grafo primale ornato con il **matching diagram** (in rosso), che indica le tuple accettate per ogni vincolo, e applichiamo AC-3, seguendone l'evoluzione: innanzitutto viene inizializzata la coda

$$Q = \{(x, y), (y, x), (z, x), (x, z), (z, w), (w, z), (w, y), (y, w)\}$$



e ora viene ripetuta l'operazione di REVISE (per ogni chiamata marchiamo in viola il valore eliminato dal dominio corrispondente)

- ① REVISE((x)y) → $D_x = \{0, 1\}$ $Q.ADD((z, x))$ ma c'è già, non viene aggiunto;
- ② REVISE((y)x) → $D_y = \{1, 2\}$ $Q.ADD((w, y))$ ma c'è già, non viene aggiunto;
- ③ REVISE((z)x) → $D_z = \{0, 1\}$ $Q.ADD((w, z))$ ma c'è già, non viene aggiunto;
- ④ REVISE((x)z) → no change;
- ⑤ REVISE((z)w) → no change;
- ⑥ REVISE((w)z) → $D_w = \{1, 2\}$ $Q.ADD((y, w))$ ma c'è già, non viene aggiunto;
- ⑦ REVISE((w)y) → $D_w = \{1\}$ $Q.ADD((z, w))$;
- ⑧ REVISE((y)w) → $D_y = \{2\}$ $Q.ADD((x, y))$;
- ⑨ REVISE((z)w) → $D_z = \{0\}$ $Q.ADD((x, z))$;
- ⑩ REVISE((x)y) → no change;
- ⑪ REVISE((x)z) → $D_x = \{0\}$ $Q.ADD((y, x))$;
- ⑫ REVISE((y)x) → no change $Q = \{\}$, STOP;

Notiamo che ogni dominio alla fine ha almeno (e anche al massimo) un valore, quindi il problema è arc consistent, ma possiamo garantire che è anche globalmente consistente? No, perché anche se ogni dominio ha almeno un valore e ogni vincolo è binario, il grafo è ciclico, quindi non possiamo affermare che sia globalmente consistente (anche se ovviamente lo è, infatti l'unico assegnamento possibile per questa rete è $a(x, y, z, w) = (0, 2, 0, 1)$).

4.4 Tecniche di ricerca

Abbiamo visto algoritmi di inferenza che rendono le reti più *strette*, così da ottenere una rete semplificata e propagare i valori assegnati a partire da una variabile arbitraria; l'inferenza aggiunge costo esponenziale (o polinomiale per algoritmi di inferenza approssimata, non li abbiamo visti) ma può restringere lo spazio di ricerca in maniera molto significativa, togliendo valori dai domini delle variabili future. Ora dobbiamo effettuare la ricerca, il cui grosso svantaggio è che dobbiamo procedere per tentativi, tornando indietro nello spazio delle soluzioni ogni volta che troviamo un'inconsistenza con gli assegnamenti precedenti (attenzione, ad ogni assegnamento *non* vengono modificati i domini): con questo approccio naive, dato un problema con n variabili e cardinalità massima dei domini d , alla profondità ℓ si avrà un branching factor $b = (n - \ell)d$ e quindi nell'albero degli stati si avranno $n!d^n$ foglie, cioè una complessità temporale e spaziale $O(n!d^n)$, inaccettabile.

Algorithm 13 Backtracking search

Esempio

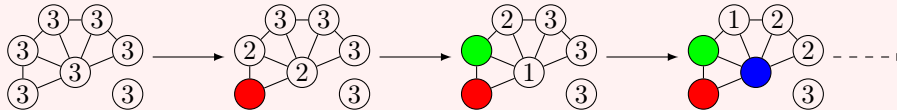
- *ordinamento* delle variabili e dei loro valori;
- *look-ahead*, ovvero predizione di inconsistenze future, e *look-back*, ovvero ritorno all'effettivo punto cruciale che ha generato l'inconsistenza;
- forzatura della *consistenza locale* (arc o path consistency sono sufficienti);
- *ristrutturare* il problema in modo che sia aciclico.

Nel caso in cui più variabili all'inizio abbiano lo stesso numero di valori nel dominio, possiamo scegliere una variabile a caso tra quelle oppure usare un tie-breaker per MRV: con la ***Degree Heuristic*** scegliamo la variabile

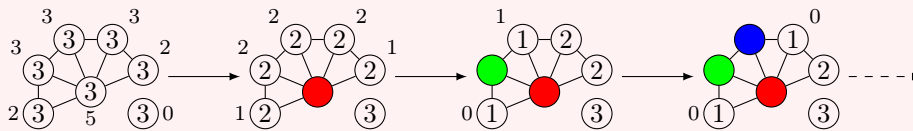
con più vincoli sulle rimanenti variabili ancora da assegnare, così da saturare rapidamente tutti i vincoli e fallire appena possibile.

Esempio

In questo grafo ogni nodo contiene il numero di valori possibili del suo dominio, che è $\{R, G, B\}$. Usando il criterio *Minimum Remaining Values*, dato che all'inizio tutti hanno lo stesso numero, viene scelto arbitrariamente il nodo a sinistra e sempre arbitrariamente il valore R , impedendo ai nodi a lui collegati di essere a loro volta rossi (ricordiamo che R rimane nel loro dominio); ora di nuovo arbitrariamente scegliamo uno tra i nodi con il dominio più piccolo, in particolare quello in alto a sinistra, il quale impedisce al nodo centrale di avere valore G , che al suo turno riceverà l'unico possibile valore, B .



Notiamo che in questo particolare caso gli assegnamenti arbitrari ci hanno permesso di arrivare comunque ad una soluzione consistente, ma questo è un caso fortunato, solitamente è necessario fare più volte backtracking. Ora rivediamo l'esempio usando anche la *Degree Heuristic*, e segniamo all'esterno dei nodi con ugual numero di valori possibili il numero di vincoli con variabili libere a cui quel nodo partecipa.



Notiamo come il nodo critico (quello centrale) venga gestito senza problemi (in questo caso ovviamente, è un'euristica, non possiamo garantire una ricerca *backtrack free*).

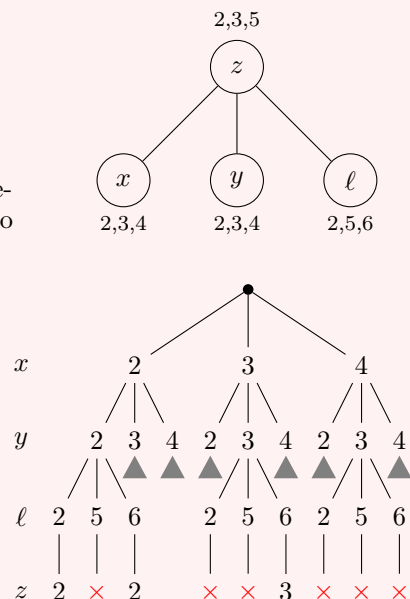
Esempio

Consideriamo la rete

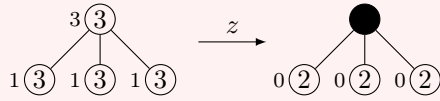
$$\begin{aligned} X &= \{x, y, \ell, z\} \\ D_{x,y} &= \{2, 3, 4\}, D_\ell = \{2, 5, 6\}, D_z = \{2, 3, 5\} \\ C &= z \text{ divide } x, y, \ell \end{aligned}$$

disegniamone il grafo primale ed esploriamo ora l'albero degli assegnamenti usando l'ordine arbitrario x, y, ℓ, z , quindi il primo livello dell'albero sarà per x , il secondo per y , ecc.

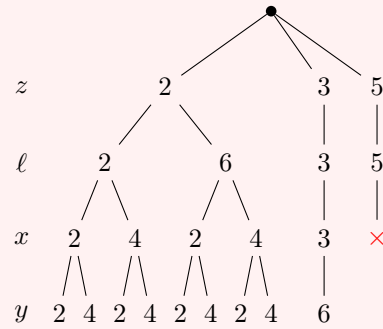
Alcuni nodi non sono stati espansi per praticità ma l'effetto è chiaro comunque: mettere l'unica variabile con dei vincoli, z , in fondo, non è affatto efficiente, in quanto ci obbliga ad espandere molti cammini fino in fondo prima di poter dire se essi siano validi o meno. Utilizzando invece un ordinamento utile delle variabili, in particolare usando MRV e la Degree Heuristic, possiamo ottenere risultati nettamente migliori, poiché verranno bloccati tutti i cammini inutili già all'inizio.



Usiamo il grafo primale per mostrare per ogni nodo il numero di valori possibili del suo dominio e il numero di vincoli con variabili libere a cui partecipa



e notiamo subito come questa euristica si confermi nuovamente vincente, infatti ci dice di far partire al ricerca dal nodo z , così da sfruttarne i vincoli fin da subito ed evitare tutti gli assegnamenti non consistenti, e poi passare agli altri nodi in ordine arbitrario ℓ, x, y (hanno tutti pari valori di MVR e DH dopo l'assegnamento di z), creando cammini *backtrack free* (caso fortunato).

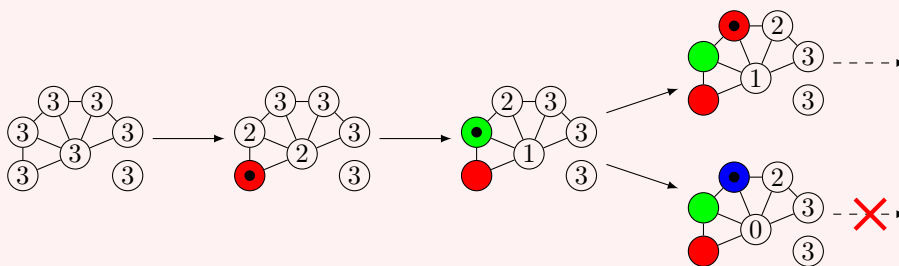


Notiamo che manca ancora qualcosa: dopo aver scelto tramite Minimum Remaining Values e Degree Heuristic la variabile migliore da esplorare, i valori vengono ancora assegnati in maniera arbitraria, senza considerare l'influenza che avranno sulle altre variabili tramite i vincoli, e questo alle volte può far espandere alberi del tutto inutili. Utilizziamo quindi la tecnica del ***Least Constraining Value***, che data una variabile ne sceglie il valore che influisce sul minor numero possibile di valori dagli altri domini e che ovviamente non li svuota, in modo da concedere più possibilità di scelta durante l'assegnamento delle prossime variabili (questo è un approccio molto simile al *look-ahead*, che vedremo tra poco).

Combinando queste tre euristiche (Minimum Remaining Values e Degree Heuristic per la scelta delle variabili e Least Constraining Value per la scelta dei valori) riusciamo a risolvere problemi di dimensioni molto significative, nell'ordine del problema delle 1000 regine.

Esempio

Dato il seguente grafo, rappresentiamo ancora all'interno di ogni nodo il numero di valori possibili del suo dominio, di nuovo $\{R, G, B\}$. L'ordine delle variabili è arbitrario (quindi non usiamo MVR e DH) e segue il semicerchio a partire da sinistra; usiamo la notazione \bullet per indicare la situazione del grafo se venisse scelto **COLOR** per il nodo corrente e, dato che per qualsiasi valore $\{R, G, B\}$ otteniamo lo stesso effetto sui domini adiacenti, scegliamo arbitrariamente **R** (sotto viene mostrato solo lo svolgimento per le nostre scelte arbitrarie, ovviamente potevamo scegliere invece **G** o **B**, ma non rappresentiamo quei cammini qui); seguiamo l'ordinamento e notiamo che anche qui qualsiasi valore va bene tranne **R** ovviamente, quindi scegliamo arbitrariamente **G**; ora arriva il problema: scegliendo **B** notiamo che il nodo centrale rimane con il dominio vuoto a causa dei vincoli, quindi dobbiamo scegliere **R**.



4.6 Considerazioni di costo

Come abbiamo già detto, forzare la consistenza per semplificare la rete permette di ridurre notevolmente i tempi della successiva ricerca, ma aggiungere nuovi vincoli aumenta il numero di controlli da effettuare sui vincoli: in una rete con soli vincoli binari, dopo la forzatura riusciremo comunque ad avere $O(n)$ controlli, ma per una rete con vincoli r -ari potremmo arrivare a $O(n^{r-1})$ controlli, che sono un costo aggiuntivo non banale. La questione rimane quindi sempre un trade-off tra forzare molto la consistenza, ed avere meno cammini da percorrere ma più controlli da svolgere a causa dei molti vincoli, o forzare poco la consistenza, ed avere meno controlli ma più cammini da percorrere; la valutazione di quale sia la scelta migliore da fare dipende dal problema e necessita di uno studio del dominio (in modo automatico oppure manualmente) e un'analisi empirica delle varie configurazioni, per capire tramite casi di esempio quale sarà la migliore da utilizzare sui casi reali.

4.7 Look-ahead

Ricapitolando, adottiamo delle tecniche per ridurre in numero totale di stati e cercare lo stato di goal in modo il più lineare possibile: idealmente queste operazioni ci danno una rete **backtrack free**, nella quale ogni nodo foglia dell'albero di ricerca è uno stato di goal, il che permette una ricerca appunto $O(n)$ sul numero delle variabili. Questo *idealmente* appunto, infatti avremo quasi sempre un qualche percorso che obbligherà il backtracking, quindi vogliamo avere una certa euristica che ci dice se un certo percorso porterà o meno ad una situazione di backtracking: le tecniche di **look-ahead** permettono di “simulare le conseguenze” della scelta di un particolare valore per la variabile corrente, permettendoci di evitare quelli che ci porteranno a delle inconsistenze. Abbiamo due strategie:

4.7.1 Forward Checking look-ahead

Data una variabile da assegnare, per ogni valore nel dominio si propagano gli effetti sui vincoli alle variabili ancora libere *separatamente*, cioè solo su quelle variabili, senza continuare con la propagazione (come si fa invece nella arc consistency) e se un dominio rimane vuoto, quel valore non potrà essere accettato. Precedentemente, come abbiamo più volte sottolineato, ad ogni assegnamento i domini non cambiavano, obbligandoci a rifare sempre i controlli sui vincoli; ora invece ad ogni assegnamento riduciamo effettivamente i domini, così da poter evitare subito i valori che produrranno domini vuoti.

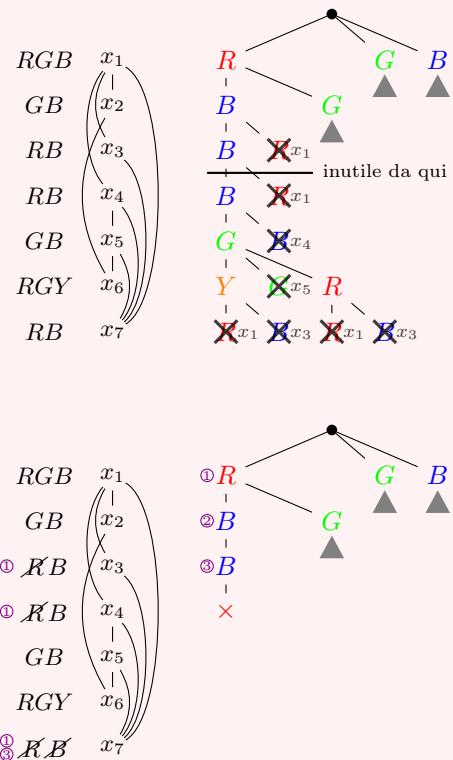
Esempio

Dato il problema di colorazione di un grafo

$$\begin{aligned} X &= \{x_1, \dots, x_7\} \\ D_{x_1} &= \{R, G, B\}, D_{x_2, x_5} = \{G, B\}, \\ D_{x_6} &= \{R, G, Y\}, D_{x_3, x_4, x_7} = \{R, B\} \\ C &= \{x_1 \neq x_2, x_1 \neq x_3, x_1 \neq x_4, x_1 \neq x_7, \\ &\quad x_2 \neq x_6, x_3 \neq x_7, x_4 \neq x_5, x_4 \neq x_7, \\ &\quad x_5 \neq x_6, x_5 \neq x_7\} \end{aligned}$$

disegniamone il grafo primale in verticale, con i domini affiancati, ed esploriamo l'albero degli assegnamenti usando l'ordine arbitrario $x_1 \dots x_7$ (quindi senza alcuna euristica). Concentriamoci ora in particolare sul ramo per $\{x_1 = R, x_2 = B\}$ (ignorando gli altri sottoalberi) e notiamo che ora x_3 può prendere solo valore B a causa del vincolo con x_1 (segniamo con \times_{x_i} l'impossibilità di assegnare un valore a causa di un vincolo con x_i), e assegnando $x_3 = B$ per x_7 non ci sarà più alcun valore possibile, ma l'algoritmo non lo può sapere finché non farà i controlli sui vincoli durante l'assegnamento di x_7 , in fondo all'albero.

Utilizzando ora il Forward Checking look-ahead, i domini vengono aggiornati dopo ogni assegnamento e notiamo che, raggiunto $x_3 = B$, unico assegnamento possibile, x_7 ha dominio vuoto, e possiamo già fermarci.



Per ogni variabile x_i abbiamo $|E_i|$ controlli di consistenza (uno per ogni vincolo a cui partecipa) per ogni coppia tra i suoi valori e quelli dell'altra variabile nel vincolo, quindi $O(|E_i||D|^2)$; questo viene ripetuto per ogni variabile, quindi dato che $\sum_i |E_i| = |E|$, il costo del Forward Checking è $O(|E||D|^2)$.

4.7.2 Arc Consistency look-ahead

Data una variabile da assegnare, per ogni valore nel dominio viene questa volta propagata integralmente la arc consistency (non *separatamente* come prima, quindi la chiameremo anche *full arc consistency*), così da applicare ricorsivamente gli effetti del valore; in questo modo possiamo osservare gli effetti di un assegnamento “un po’ più in là” rispetto alla strategia Forward Checking. Per applicare la arc consistency useremo l'algoritmo AC-3, ma sarebbe ancora meglio usare AC-4 (che non abbiamo visto) poiché ha costo $O(|E||D|^2)$.

Esempio

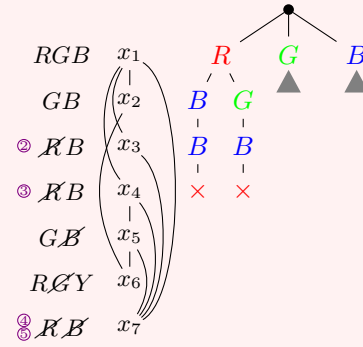
Usando il problema precedente, sempre disegnandone il grafo primale in verticale, con i domini affiancati, usiamo ora la Arc Consistency look-ahead, ed esploriamo l'albero degli assegnamenti usando ancora l'ordine arbitrario $x_1 \dots x_7$. Per x_1 concentriamoci sul ramo per R (ignorando nuovamente i sottoalberi per le altre assegnazioni), il quale appunto ora forzerà la arc consistency: inizializziamo la coda con i vincoli a cui partecipa x_1 (non inseriamo e tuple (x_1, \cdot) semplicemente perché x_1 non è più una variabile libera, le stiamo assegnando R)

$$Q = \{(x_2, x_1), (x_3, x_1), (x_4, x_1), (x_7, x_1)\}$$

e ora seguiamo passo passo i vari REVISE

- ① REVISE($(x_2)x_1$) \rightarrow no change;
- ② REVISE($(x_3)x_1$) $\rightarrow D_{x_3} = \{B\}$ $Q.ADD((x_7, x_3))$;
- ③ REVISE($(x_4)x_1$) $\rightarrow D_{x_4} = \{B\}$ $Q.ADD((x_5, x_4), (x_7, x_4))$;
- ④ REVISE($(x_7)x_1$) $\rightarrow D_{x_7} = \{B\}$ $Q.ADD((x_3, x_7), (x_4, x_7), (x_5, x_7))$;
- ⑤ REVISE($(x_7)x_3$) $\rightarrow D_{x_7} = \{\}$ STOP;

Notiamo che già a $x_1 = R$ sappiamo che x_7 non avrà valori per qualsiasi valore delle altre variabili, quindi possiamo evitare di esplorare questo sottoalbero.



Notiamo che ovviamente questa strategia è più costosa di Forward Checking, e infatti usando AC-3 il costo è $O(|E||D|^3)$, quindi è bene considerare anche questo trade-off in fase di progettazione: usare Arc Consistency look-ahead taglia dei livelli in più dell'albero di ricerca ma è cubica nei domini, dunque sarebbe meglio usarla quando abbiamo molte variabili ma domini molto limitati (se usassimo AC-4 il costo sarebbe uguale).

Esiste una variante della Arc Consistency look-ahead chiamata *Maintaining Arc Consistency look-ahead*, la quale utilizza lo stesso procedimento di Arc Consistency look-ahead per rigettare i valori, ma ora applica la arc consistency su tutta la rete ad ogni rigetto; in questo modo la rete viene resa sempre snella ad ogni riduzione di dominio e si ottengono dei risultati ancora migliori, ma non lo vedremo.

4.8 Ristrutturazione del problema

In una osservazione abbiamo visto che per reti a vincoli acicliche il costo di ricerca è polinomiale, quindi vogliamo trasformare la rete per ottenere una struttura aciclica, così da forzare le condizioni per avere un *caso trattabile*. Ci sono principalmente due tecniche di ristrutturazione: *rimuovere* variabili dal problema o *raggrupparle*.

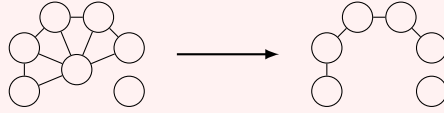
4.8.1 Cycle conditioning

Dato un grafo non diretto, un *cycle subset* è un sottoinsieme dei suoi nodi che se *rimossi* (verrà chiamato per questo anche *cycle cutset*) producono un sottografo aciclico, cioè un albero.

L'idea è quindi di rimuovere dalla rete ogni variabile dopo che essa è stata assegnata, facendo cioè *conditioning*: in questo modo otterremo una sottorete il cui costo per trovare la soluzione sarà comunque esponenziale (devo comunque propagare la consistenza e fare ricerca), ma sulla dimensione del cycle cutset (devo farlo per ogni configurazione delle variabili nel cutset), quindi avremo un costo molto basso (adottando l'algoritmo giusto) per dimensioni piccole del cutset; l'unico problema è trovare rapidamente un cycle cutset, che essendo un problema NP-completo avviene tramite delle euristiche, che vedremo alla fine.

Esempio

Usando il nostro solito esempio, il cycle cutset contiene solo il nodo centrale, che notiamo essere l'unica causa di tutti cicli nel grafo: il cutset (in pratica solo il nodo centrale) viene rimosso e otteniamo un albero (in questo caso una *foresta*, cioè più alberi separati), sul quale andiamo ora ad eseguire AC-3 per ogni configurazione del cutset rimosso (in questo caso, ogni valore $\{R, G, B\}$).



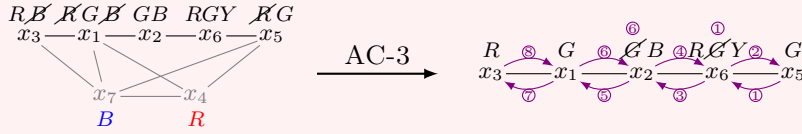
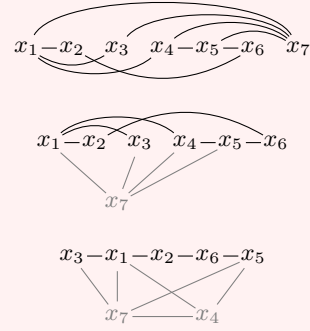
Notiamo che in questo caso capiamo intuitivamente qual è il cutset, ma dobbiamo trovare un algoritmo per farlo in modo automatico.

Esempio

Dato il problema di colorazione già visto sopra, disegniamone il grafo primale e costruiamo il cutset con un'euristica arbitraria, cioè togliendo x_7 ed x_4 .

Andiamo a vedere come cambia il grafo primale togliendo queste variabili una alla volta (l'algoritmo effettivo le toglie insieme, questo è solo un esempio), mantenendo comunque traccia del loro vincolo con le altre variabili.

Ora dobbiamo applicare arc consistency ad ogni configurazione tra x_4 e x_7 , cioè $D_4 \times D_7 = \{R, B\} \times \{R, B\}$; notiamo che le configurazioni (R, R) e (B, B) violano il vincolo $x_4 \neq x_7$ e dunque al loro turno verranno immediatamente scartate: questo ci permette di applicare effettivamente arc consistency solo su (R, B) e (B, R) . Vediamo solo il procedimento per $x_4 = R, x_7 = B$: prima propaghiamo i vincoli degli assegnamenti e subito dopo applichiamo AC-3, come specificato dall'algoritmo, arbitrariamente a partire da x_5 , segnando le direzioni dei REVISE.



Notiamo che per $x_4 = R, x_7 = B$ la rete è arc consistent, quindi per questo assegnamento parziale le altre variabili hanno almeno un valore e il problema è risolvibile con le soluzioni $a(x_1 \dots x_7) = (G, B, R, R, G, \{R \vee Y\}, B)$; se non fosse stata arc consistent avremmo dovuto ripetere tutto per la prossima configurazione, cioè $x_4 = B, x_7 = R$.

4.8.1.1 Tree decomposition

Possiamo dimostrare che un qualsiasi grafo ciclico è sempre *compilabile* in una struttura simil-albero aciclica, ad un costo che dipende dalla sua *struttura topologica*; anziché parlare di cycle cutset in termini di variabili, useremo le relazioni tra esse, sfruttando gli *ipergrafi*.

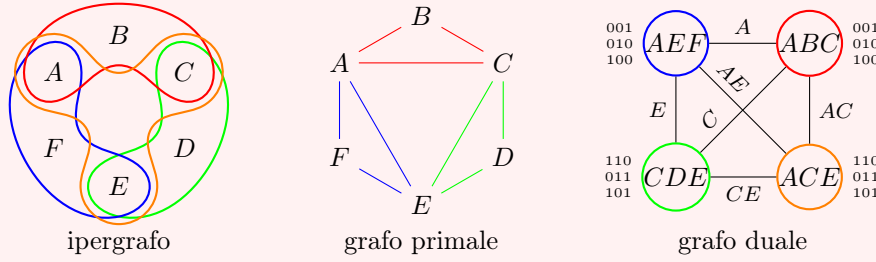
Per la trasformazione di un grafo in un albero, la rete a vincoli viene ora espressa tramite un *ipergrafo*, che rappresenta sottoinsiemi di variabili e le relazioni tra questi; formalmente, un ipergrafo $H = (V, S)$ è formato da degli ipervertici $V = \{v_1 \dots v_n\}$ e degli iperarchi $S = \{S_1 \dots S_r\}$ dove $S_i \subseteq V$.

In generale, ogni rete a vincoli è rappresentabile come un ipergrafo, quindi data una rete a vincoli $\mathcal{R} = (X, D, C)$, con vincoli $C = \{R_{S_1} \dots R_{S_r}\}$ di scope S_i , avremo

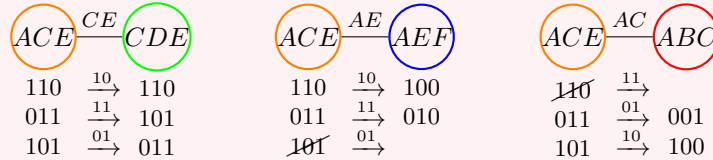
- l'ipergrafo $\mathcal{H}_{\mathcal{R}} = (X, S)$ con $S = \{S_1 \dots S_r\}$;
 - il grafo primale $\mathcal{H}_{\mathcal{R}}^p = (X, Q)$ dove Q è l'insieme di tutti gli archi possibili per ogni S_i ;
 - il grafo duale $\mathcal{H}_{\mathcal{R}}^d = (S, E)$ dove $(S_i, S_j) \in E \iff S_i \cap S_j \neq \{\}$, cioè se hanno delle variabili in comune;
- Notiamo che per reti a vincoli binari, l'ipergrafo di quella rete è identico al grafo primale.

Esempio

Data una rete a vincoli, rappresentiamone l'ipergrafo, con $V = \{A, B, C, D, E, F\}$ e $S = \{\{A, E, F\}, \{A, B, C\}, \{C, D, E\}, \{A, C, E\}\}$, il grafo primale e il grafo duale, notando che nel grafo primale gli archi per il vincolo ACE sono già presenti a causa degli altri vincoli; nel duale rappresentiamo inoltre le tuple accettate da ogni vincolo.



Intuitivamente (per adesso) espandendo i vincoli a partire da ACE notiamo che alcune tuple non hanno corrispettivi negli altri vincoli.



Dato che solo $ACE = 011$ ha corrispettivi per tutti gli altri vincoli, ne espandiamo i valori e otteniamo al soluzione $ABCDEF = 001010$.

Vediamo più formalmente quello che è stato fatto intuitivamente nell'esempio: dato un ipergrafo con grafo duale $G = (V, E)$, eliminiamo i suoi vincoli ridondanti cercando di ottenere un **sottografo degli archi** $G' = (V, E')$ con $E' \subseteq E$ (in pratica è G senza alcuni archi) che rispetti la **Running Intersection Property** (o **connectedness**), ovvero tale che per ogni coppia di nodi che condividono una variabile esiste un cammino formato da archi etichettati con quella variabile (in pratica è un sottografo che non contiene gli archi ottenibili in modo equivalente tramite dei percorsi); se questo sottografo esiste è detto il **join graph** del grafo duale G , e nel caso fosse aciclico, il **join tree**.

Con queste definizioni possiamo dire che una rete a vincoli è **aciclica** (e quindi risolvibile in tempo polinomiale, che è il nostro obiettivo) se il suo ipergrafo è in realtà un **iperalbero**, cioè un ipergrafo il cui grafo duale è esprimibile con un **join tree**.

Esempio

Per il problema appena visto, intuitivamente notiamo che nel grafo duale l'arco (AEF, CDE) etichettato E può essere eliminato perché esiste il percorso $(AEF, ACE), (ACE, CDE)$ che contiene in ogni arco la variabile E ; lo stesso vale per gli archi (AEF, ABC) e (CDE, ABC) . Il **sottografo degli archi** che otteniamo eliminando gli archi $\{(AEF, CDE), (AEF, ABC), (CDE, ABC)\}$ rispetta la running intersection property, ed è quindi un **join tree** del grafo duale.



Dato che il grafo duale è esprimibile con un join tree, la rete è aciclica. Per adesso questo procedimento è totalmente intuitivo e manuale, va vedremo degli algoritmi per automatizzarlo.

Poniamoci nel caso in cui il join tree sia stato già trovato (vedremo poi come farlo): possiamo ora percorrerlo verificando la **consistenza diretta**, cioè la versione unidirezionale della arc consistency, che in questo caso è ovviamente più efficiente rispetto alla arc consistency poiché ora abbiamo un albero e non un grafo, quindi la direzione in cui propagare è una sola.

Algorithm 14 TREE-SOLVER**Require:** una rete a vincoli aciclica \mathcal{R} , un join tree T per \mathcal{R} **Ensure:** soluzione o inconsistenza

```

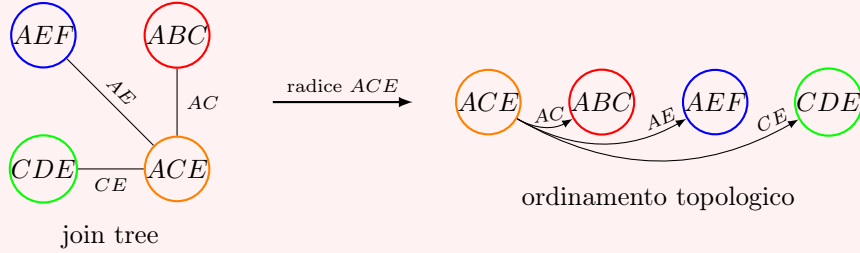
1: // usiamo le relazioni perché stiamo lavorando sul grafo duale
2: genera  $d = \{R_1 \dots R_r\}$  ordine topologico (dalla radice alle foglie) indotto da  $T$ 
3: // forzo la consistenza diretta dalle foglie alla radice
4: for all  $j = r$  to 1 do
5:   for all arco  $(j, k) \in T$  dove  $k < j$  do
6:      $R_k \leftarrow \pi_{S_k}(R_k \bowtie R_j)$  // è REVISE( $(x_k)x_j$ )
7:   if  $R_k = \{\}$  then return no solution
8: // propaghiamo le tuple della radice
9: seleziona una tupla in  $R_1$ 
10: for all  $i = 2$  to  $r$  do
11:   seleziona una tupla consistente con tutte le tuple  $R_1 \dots R_{i-1}$  già assegnate
12: return la soluzione

```

In pratica, prima vengono ordinate le relazioni in *ordine topologico* dalla radice alle foglie (ogni foglia deve essere ordinata dopo la sua radice, si ottiene scorrendo l'albero come fa la BFS ad esempio), e poi viene forzata la consistenza diretta dalle foglie alla radice; infine, dalla radice si espandono le tuple una a una finché non si trova la prima soluzione consistente.

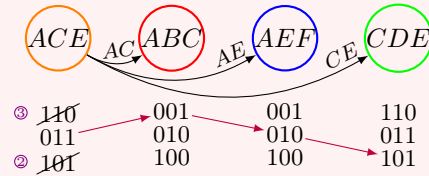
Esempio

Dato il join tree di un grafo duale, scegliamo ACE come nodo radice, ordiniamo i nodi topologicamente (in ordine di visita secondo la BFS)



e forziamo la consistenza diretta dalle foglie alle radici, quindi seguendo l'ordine topologico al contrario, e mostriamo ogni passo $R_k \leftarrow \pi_{S_k}(R_k \bowtie R_j)$

- ① $R'_{ACE} \leftarrow \pi_{ACE}(R_{ACE} \bowtie R_{CDE});$
- ② $R''_{ACE} \leftarrow \pi_{ACE}(R'_{ACE} \bowtie R_{AEF});$
- ③ $R'''_{ACE} \leftarrow \pi_{ACE}(R''_{ACE} \bowtie R_{ABC});$



Dato che nessun dominio è rimasto vuoto avremo sicuramente almeno una soluzione, quindi espandiamo i vincoli all'interno dell'albero per ogni tupla rimasta ad ACE , cioè solo 011, e otteniamo la soluzione $ABCDEF = 001010$.

Notiamo che alcune tuple, ad esempio $ABC = 010$, pur non avendo un corrispettivo con la radice non sono state tolte: questo avviene perché stiamo forzando la consistenza diretta, unidirezionale, e non la arc consistenza, che è bidirezionale.

Notiamo inoltre che se avessimo scelto un altro nodo come radice sarebbe cambiato tutto l'ordine topologico, ma la soluzione sarebbe stata la stessa.

Ricapitolando, una rete a vincoli può essere privata in qualche modo dei nodi che la rendono ciclica (le variabili nel cycle cutset o equivalentemente i vincoli ridondanti nel grafo duale), così da ottenere un join tree, sul quale possiamo usare TREE-SOLVER per trovare al costo $O(|D|^c(|V| - c)|D|^2)$ una soluzione della rete, se esiste. Il costo è stato calcolato nel seguente modo: per ognuna delle $|D|^c$ configurazioni del cycle cutset, formato da c variabili (che sono le stesse nei vincoli ridondanti nel grafo duale), eseguiamo $R_k \leftarrow \pi_{S_k}(R_k \bowtie R_j)$, che ha costo $|D|^2$, per ogni arco nel join tree, ovvero $|V| - c$ volte.

Come abbiamo detto all'inizio e come notiamo con evidenza ora, questo algoritmo è nettamente migliore di AC-3 per dimensioni piccole del cycle cutset.

Il problema ora rimane trovare il cycle cutset, ma non trattiamo esplicitamente questo problema; andiamo invece a vedere come è possibile verificare rapidamente se una rete a vincoli sia o meno aciclica: continuando a utilizzare gli ipergrafi come struttura, dobbiamo definire degli algoritmi che trovano rapidamente un sottografo degli archi che soddisfa la Running Intersection Property (join graph), così da poter controllare poi che tale sottografo sia un albero (join tree); per farlo, possiamo concentrarci sul grafo primale oppure sul duale.

4.8.1.2 Dual Based Recognition

Ideato da Maier nel 1983, questo algoritmo capisce se una rete a vincoli è aciclica sfruttando il fatto che se un ipergrafo ha un join tree allora un qualsiasi *maximum spanning tree* del suo grafo duale è un join tree. Il *maximum spanning tree* di un grafo è un sottografo aciclico (un albero appunto) che tocca ogni nodo passando dai percorsi più costosi (gli archi devono essere pesati) e può essere calcolato usando l'algoritmo di Kruskal. L'idea è di

1. costruire il grafo duale dall'ipergrafo della rete a vincoli;
2. impostare il peso degli archi con il numero di variabili in comune tra i vincoli di quell'arco (interpretando quindi il costo degli archi come il numero di variabili che essi espandono);
3. ottenere il maximum spanning tree del grafo duale usando l'algoritmo di Kruskal (che è lineare nel numero degli archi);
4. controllare se il maximum spanning tree (che è effettivamente un iperalbero per la rete a vincoli) è un join tree (essendo un albero c'è un solo cammino per ogni coppia di nodi e la Running Intersection Property è verificabile rapidamente).

Se il maximum spanning tree è un join tree allora la rete a vincoli è aciclica e risolvibile polinomialmente.

Algorithm 15 DUALACYCLICITY

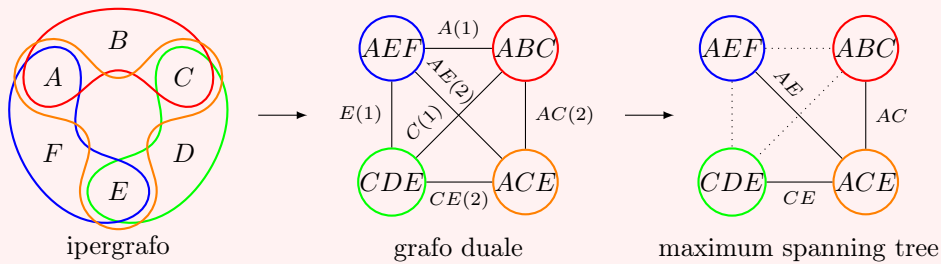
Require: un ipergrafo $H_{\mathcal{R}} = (X, S)$ per la rete a vincoli $\mathcal{R} = (X, D, C)$

Ensure: un join tree $T = (S, E)$ per $H_{\mathcal{R}}$ se \mathcal{R} è aciclica

- 1: genera il maximum spanning tree $T = (S, E)$ per il grafo duale pesato di \mathcal{R}
 - 2: // **controlla la Running Intersection Property su ogni cammino**
 - 3: **for all** coppia $u, v \in S$ **do**
 - 4: **if** RIP non è soddisfatta sull'unico cammino tra u e v **then**
 - 5: **return** \mathcal{R} non è aciclica
 - 6: **return** \mathcal{R} è aciclica e T è il suo join tree
-

Esempio

Usando la rete vista prima, andiamo ora a trovare il maximum spanning tree con l'algoritmo DUALACYCLICITY e notiamo che è lo stesso che abbiamo intuitivamente identificato all'inizio: dall'ipergrafo otteniamo il grafo duale, al quale aggiungiamo il peso tra parentesi (è il numero di variabili in comune tra i due vincoli che collega).

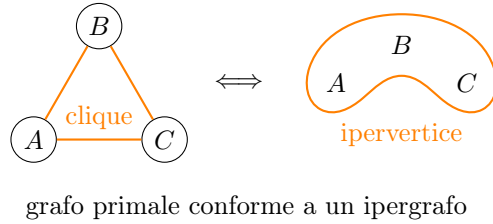
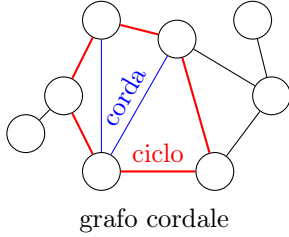


Per creare il maximum spanning tree ordiniamo gli archi per peso e li aggiungiamo all'albero finché con essi non avremo toccato tutti i nodi del grafo. A questo punto verifichiamo la Running Intersection Property sull'albero e notiamo che vale, quindi l'albero è un join tree (è esattamente lo stesso che avevamo estratto prima manualmente).

4.8.1.3 Primal Based Recognition

In questo caso per capire se una rete a vincoli è aciclica verifichiamo che il *grafo primale* sia

- **cordale**, ovvero che nel grafo primale ogni ciclo di lunghezza almeno 4 abbia una corda, cioè un arco non appartenente al ciclo che connette due vertici non adiacenti nel ciclo;
- **conforme** rispetto al suo ipergrafo, ovvero che esista un mapping 1:1 tra le *clique massimali* nel grafo primale (cioè che non sono contenute in una clique più grande) e gli ipervertici dell'ipergrafo (in pratica, se le variabili completamente connesse tra loro nel grafo primale si rispecchiano nello scope di uno ed un solo vincolo);



Entrambi questi controlli sono fattibili usando un **ordine di massima cardinalità** che, dato un primo nodo arbitrario, pone come prossimo nodo quello che ha il maggior numero di connessioni (nel grafo) coi nodi già ordinati (nel caso ce ne fosse più di uno, ne viene scelto uno arbitrariamente):

- **cordalità**: un grafo è *cordale* se in un suo qualsiasi ordine di massima cardinalità i **predecessori** di ogni nodo (nodi adiacenti ad un nodo che nell'ordine lo precedono) formano una clique (cioè sono completamente collegati tra loro);
- **conformità**: per trovare la clique massimale di un certo nodo nel grafo basta ordinare i nodi per cardinalità massima e considerare ogni nodo precedente a quel nodo.

Con queste basi l'idea per l'algoritmo è la seguente:

1. costruisci un ordine di cardinalità per il grafo primale;
2. controlla se il grafo è cordale usando l'ordine e verificando che per ogni nodo, i suoi predecessori formino una clique;
3. controlla se il grafo è conforme usando l'ordine ed estraendo le clique massimali nel grafo e confrontandole con gli scope dei vincoli.

Algorithm 16 PRIMALACYCLICITY

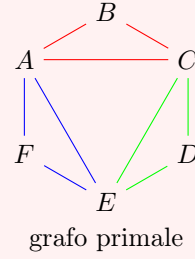
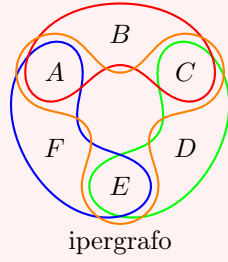
Require: un grafo primale $G_{\mathcal{R}}$ per la rete a vincoli $\mathcal{R} = (X, D, C)$

Ensure: un join tree $T = (S, E)$ per $H_{\mathcal{R}}$ se \mathcal{R} è aciclica

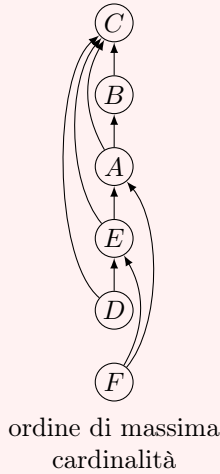
- 1: genera $d_M = \{x_1 \dots x_n\}$ ordine massimale di $G_{\mathcal{R}}$
 - 2: *// test cordalità usando d_M*
 - 3: **for all** $i = n$ to 1 **do**
 - 4: **if** predecessori di x_i non sono completamente connessi **then**
 - 5: **return** \mathcal{R} non è aciclica
 - 6: *// test conformità usando d_M*
 - 7: genera le clique massimali $\{C_1 \dots C_r\}$ usando d_M
 - 8: **for all** $i = n$ to 1 **do**
 - 9: **if** C_i è formata da tutte le variabili nello scope S_j di un vincolo $C_j \in C$ **then**
 - 10: *// per ora, \mathcal{R} è aciclica*
 - 11: **else**
 - 12: **return** \mathcal{R} non è aciclica
 - 13: *// \mathcal{R} è aciclica*
 - 14: genera il join tree T del grafo duale delle clique massimali
 - 15: **return** \mathcal{R} è aciclica e T è il suo join tree
-

Esempio

Utilizziamo il problema precedente, di cui etichettiamo per convenienza i vincoli $\{C_1 = ABC, C_2 = ACE, C_3 = CDE, C_4 = AEF\}$, e applichiamo l'algoritmo PRIMALCYCLICITY.



Creiamo inizialmente ① l'ordine di massima cardinalità arbitrariamente a partire dal nodo C , poi aggiungiamo il nodo che ha più connessioni con i nodi già ordinati, e dato che tutti i nodi A, B, D, E hanno la stessa cardinalità rispetto a $\{C\}$, selezioniamo arbitrariamente B . Ora aggiungiamo il nodo con cardinalità maggiore rispetto a $\{B, C\}$, che è A , e poi rispetto ad $\{A, B, C\}$, che è E ; infine dato che sia D sia F hanno cardinalità 2 rispetto a $\{A, B, C, E\}$, scegliamo arbitrariamente D , e poi segue in ultimo F . Mostriamo a lato l'ordine di massima cardinalità così ottenuto, e per nostra comodità segniamo con delle frecce i predecessori per ogni nodo.



Ora ② per il test della cordalità scorro l'ordine dal fondo e controllo se per ogni nodo i suoi predecessori sono completamente connessi, cioè se formano una clique (ad esempio, i predecessori di E sono A ed C , che sono connessi tra loro nel grafo primale, quindi E passa il test):

$Anc(F) = \{A, E\} \rightarrow \{A, E\}$ è una clique? sì
 $Anc(D) = \{C, E\} \rightarrow \{C, E\}$ è una clique? sì
 $Anc(E) = \{A, C\} \rightarrow \{A, C\}$ è una clique? sì
 $Anc(A) = \{B, C\} \rightarrow \{B, C\}$ è una clique? sì
 $Anc(B) = \{C\} \rightarrow \{C\}$ è una clique? sì
 $Anc(C) = \{\} \rightarrow \emptyset$ è una clique? sì

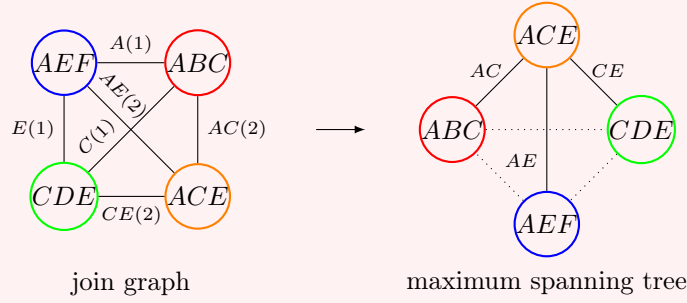
Infine, ③ per il test della conformalità scorro l'ordine dal fondo e controllo che le clique massimali *nel grafo primale* siano mappate 1:1 ai vincoli del problema.

Alle clique trovate sopra aggiungiamo il nodo stesso che le ha trovate, così da farle diventare massimali *per quel nodo* (ad esempio per E è stata trovata la clique $\{A, C\}$, che non può essere massimale in quanto E è collegato ad ogni nodo in $\{A, C\}$, quindi la clique massimale per E è $\{E, A, C\}$); fatto questo, andiamo a controllare tra queste clique se quelle massimali *nel grafo* sono mappate a dei vincoli:

$\{F, A, E\}$ è una clique massimale nel grafo e si mappa a $C_4 = AEF$
 $\{D, C, E\}$ è una clique massimale nel grafo e si mappa a $C_3 = CDE$
 $\{E, A, C\}$ è una clique massimale nel grafo e si mappa a $C_2 = ACE$
 $\{A, B, C\}$ è una clique massimale nel grafo e si mappa a $C_1 = ABC$
 $\{B, C\}$ è contenuta in $\{A, B, C\}$, quindi non è massimale nel grafo e la ignoro
 $\{C\}$ è contenuta ad esempio in $\{A, B, C\}$, come prima, la ignoro

Dato che tutte le clique massimali nel grafo sono mappate 1:1 a dei vincoli, il test della conformità è superato e quindi \mathcal{R} è una rete aciclica.

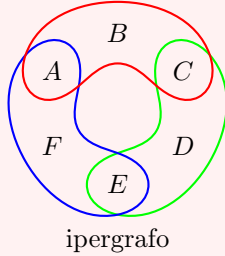
L'ultimo passaggio è ④ estrarre il maximum spanning tree T del grafo duale costruito usando le clique massimali, al quale vengono aggiunti anche i pesi sugli archi (il numero delle variabili in comune):



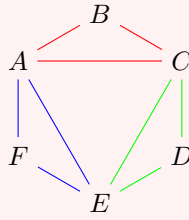
Con il join tree ora basta semplicemente eseguire TREE-SOLVER e avremo la soluzione per il problema.

Esempio

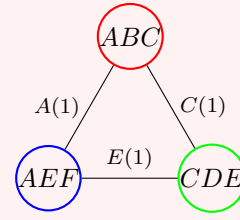
Notiamo che per una rete identica a quella precedente ma senza vincolo ACE il grafo primale è lo stesso, e di conseguenza il test di cordalità estrae le stesse identiche clique, ma il test di conformità fallisce, in quanto per la clique $\{E, A, C\}$ non si riesce a mappare nessun vincolo, quindi la rete non è aciclica. Per questo problema alternativo quindi PRIMALACYCLICITY fallisce sul mapping, ma vediamo come si comporta invece DUALACYCLICITY.



ipergrafo

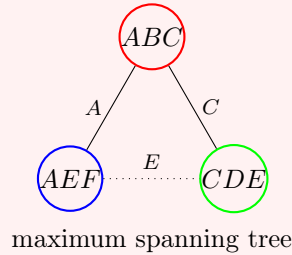


grafo primale



grafo duale

Estraiamo un maximum spanning tree dal grafo duale e, dato che tutti gli archi hanno cardinalità uguale, scegliamo arbitrariamente ABC come nodo radice.



maximum spanning tree

Notiamo che per questo maximum spanning tree (e in particolare per il teorema di Maier, per qualsiasi maximum spanning tree) non vale la Running Intersection Property, infatti il cammino da AEF a ABC contiene A per tutte le etichette, il cammino da CDE a ABC contiene C per tutte le etichette, ma il cammino da AEF a CDE non contiene E per tutte le etichette e in particolare in nessuna di esse; questo significa che potremmo trovare delle tuple in accordo tra AEF e ABC e tra CDE e ABC che però creano un conflitto tra AEF e CDE . In conclusione, non esiste un maximum spanning tree che è un join tree per il grafo duale, quindi la rete non è aciclica, come detto all'inizio.

4.8.2 Clustering

Abbiamo visto che quando una rete è aciclica (lo controlliamo con i metodi visti sopra) possiamo risolverla in modo efficiente con TREE-SOLVER; quando una rete *non* è aciclica troviamo il cycle cutset e, per ognuna delle configurazioni possibili del cycle cutset, applichiamo TREE-SOLVER sulla rete privata delle variabili nel cycle cutset. Abbiamo detto che trovare il cycle cutset è NP-completo, quindi abbiamo bisogno di un modo alternativo per gestire questa problematica: diamo un algoritmo per *compilare* una qualsiasi rete ciclica in una struttura aciclica, per poi al solito usare TREE-SOLVER e risolverla efficientemente.

Con il **clustering** creo gruppi di vincoli e li uso per creare una struttura simil-albero in cui i nodi sono dei sottoproblemi, risolvo singolarmente ogni sottoproblema (in pratica sfruttiamo il *divide et impera* e questo è il

collo di bottiglia, più piccoli sono i problemi meglio è) e infine accorpo le soluzioni consistenti tra loro usando TREE-SOLVER. Ci sono due metodologie che possiamo adottare: *Join Tree Clustering* oppure *Cluster Tree Elimination* (che non trattiamo).

4.8.2.1 Join Tree Clustering

L'algoritmo è una piccola modifica del primal based recognition, nel quale durante il test della cordalità, se un nodo non dovesse appartenere ad una clique, lo forziamo ad averne una, *inducendo* archi tra i suoi predecessori laddove dovessero mancare. L'idea è la seguente:

1. scelgo un ordine delle variabili;
2. creo il *grafo indotto* del grafo primale utilizzando l'ordine, andando a forzare la cordalità e garantendo quindi la Running Intersection Property;
3. crea il join tree usando un maximum spanning tree sulle clique massimali nel *grafo indotto* (i pesi sono il numero delle variabili in comune, come sempre). Questo è possibile poiché estraendo le clique massimali ottengo l'ipergrafo del grafo indotto (i cui ipervertici sono i vincoli, che sono appunto le clique), dal quale posso estrarre il grafo duale, sempre del grafo indotto (che ha come vertici gli scope dei vincoli), sul quale posso tranquillamente usare Kruscal per estrarre il join tree;
4. alloco almeno un vincolo ad ogni clique (vertice del join tree) che ne contiene lo scope, cioè codifico ogni sottoproblema (che è la soddisfazione di un vincolo) usando le clique. In pratica stiamo verificando la conformalità del grafo indotto rispetto al suo ipergrafo;
5. trovo le tuple accettate da ogni clique (cioè risolvo ogni sottoproblema);
6. applico TREE-SOLVER sul join tree, così da associare le tuple consistenti tra loro e risolvere la rete.

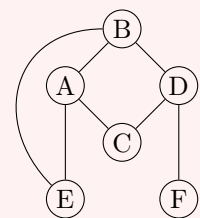
Il *grafo indotto* G^* di un grafo G secondo un ordine d è ottenuto scorrendo dal fondo l'ordine e aggiungendo al grafo tutti gli archi necessari affinché ogni nodo nell'ordine abbia i suoi predecessori completamente connessi tra loro (in pratica, forzo le clique tra i predecessori di ogni nodo a partire dal fondo dell'ordine). Questo fa sì che per costruzione l'ordine d sia un'**ordine di eliminazione perfetto**, ovvero un ordine tale per cui per ogni nodo v nell'ordine, v e i suoi predecessori formano una clique, e questo è essenziale per la cordalità, infatti Fulkerson e Gross hanno dimostrato che un grafo è cordale se e solo se ha un'ordine di eliminazione perfetto. Proprio per questo teorema, il grafo indotto di un grafo è cordale per costruzione.

Abbiamo detto che più piccoli sono i sottoproblemi, meglio è, e dato che effettivamente stiamo parlando di clique, il parametro da considerare è il numero di nodi di cui sono composte. Dato che le clique vengono estratte dai predecessori di ogni nodo seguendo l'ordine, definiamo **ampiezza** w del nodo v il numero dei suoi predecessori e definiamo l'**ampiezza di un grafo** come l'ampiezza massima tra i suoi nodi; durante il calcolo della complessità di questo algoritmo useremo l'ampiezza come parametro determinante.

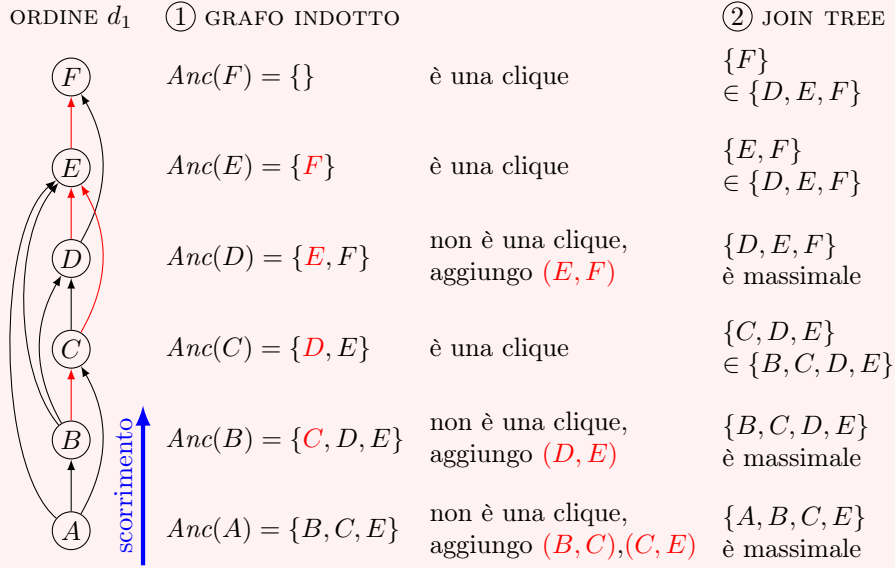
Esempio

Data la rete a vincoli qui a lato, ci concentriamo solo sui passaggi fino a TREE-SOLVER, quindi i domini non ci interessano e nemmeno i vincoli in sé. Utilizziamo arbitrariamente l'ordine $d_1 = \{F, E, D, C, B, A\}$ per costruire il grafo indotto (i passaggi sono identici a PRIMALCYCLICITY fino al test della cordalità, nel quale ora aggiungo gli archi mancanti); mostriamo l'ordinamento, indicando con delle frecce i nodi predecessori.

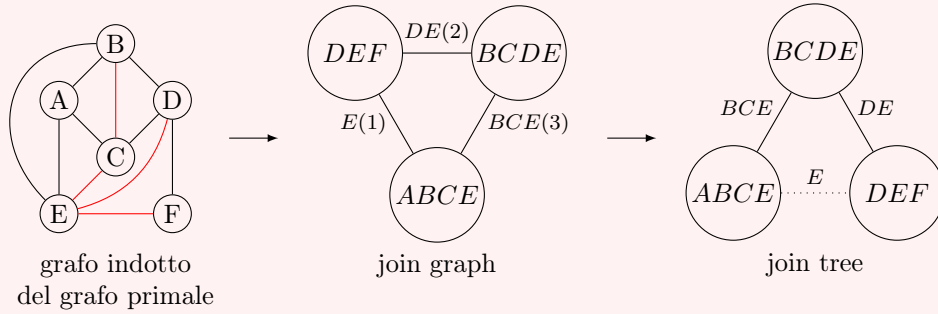
Ora ① costruiamo il grafo indotto estraendo le clique partendo dal fondo, come abbiamo fatto per il test della cordalità, e completiamo le clique incomplete, facendo attenzione a considerare per i nodi successivi anche tutti gli archi appena aggiunti (il grafo indotto appena creato viene mostrato di seguito). Poi, ② costruisco il join graph del grafo indotto e successivamente ne estraggo il join tree (che ricordiamo essere semplicemente il suo maximum spanning tree): questo avviene scorrendo l'ordine dal fondo ed estraendo le clique massimali *nel grafo indotto*, ricordando che in pratica sono le clique appena trovate unite al nodo che le ha trovate, che poi vanno controllate tra loro.



grafo primale



Vediamo il risultato dei passaggi sopra: la forzatura degli archi nelle clique ha appunto creato il grafo indotto, dal quale abbiamo estratto direttamente il join graph (usando le clique massimali) e successivamente il join tree (che ne è il maximum spanning tree).



Ora ③ controllo che almeno un vincolo sia contenuto in ogni clique; utilizziamo la notazione R_{uv} per scrivere in maniera compatta il vincolo (u, v) :

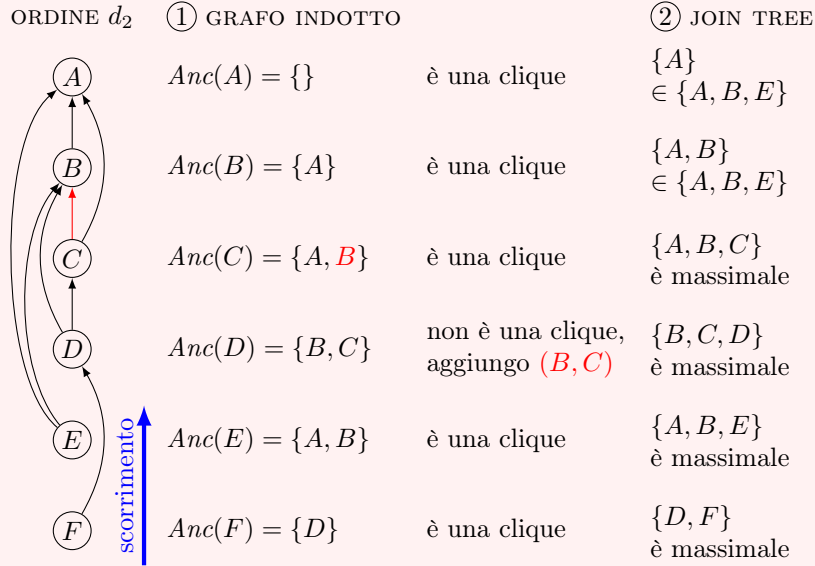
$\{A, B, C, E\}$ contiene $R_{AB}, R_{AC}, R_{AE}, \underline{R_{BE}}$ (*arbitrario*)
 $\{B, C, D, E\}$ contiene R_{BD}, R_{CD}
 $\{D, E, F\}$ contiene R_{DF}

Dato che tutte le clique massimali nel grafo contengono dei vincoli e ogni vincolo è contenuto in una clique, \mathcal{R} è una rete aciclica: ogni vincolo è risolvibile localmente da una clique forzando la consistenza dei domini e unisco i risultati (ovvero le tuple accettate dalle clique, che codificano la soddisfacibilità dei vincoli al loro interno) usando TREE-SOLVER per trovare la soluzione.

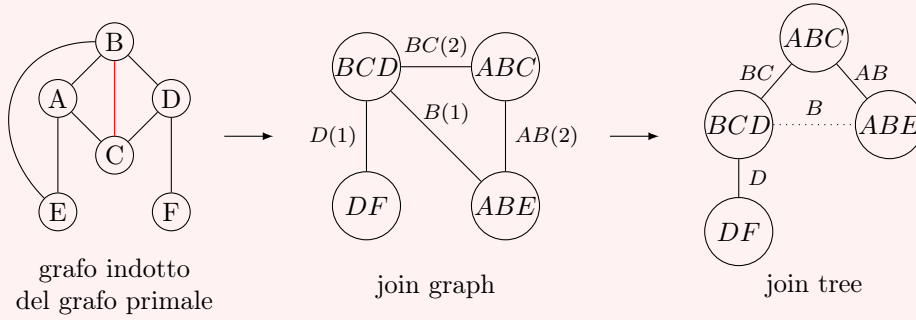
La complessità del Join Tree Clustering è dominata dall'estrazione delle soluzioni dei sottoproblemi, che è esponenziale nella grandezza delle clique, e in particolare dalla dimensione della clique massima (quella con più vertici); tale clique ha dimensione pari all'ampiezza del grafo indotto $w^* + 1$ per definizione, e dato che l'ampiezza è completamente determinata dall'ordine scelto, l'ordine è un parametro cruciale per l'algoritmo: purtroppo trovare l'ordine ottimo è un problema NP-completo, e in generale quindi si usano delle euristiche trovarne uno ottimale. In particolare, l'ordine di massima cardinalità è un'ottima euristica perché, anche se non sempre darà casi ottimi, sicuramente non aggiungerà mai archi inutili se il grafo è già cordale e quindi le clique saranno delle dimensioni minori possibile.

Esempio

Applichiamo nuovamente l'algoritmo sulla stessa rete a vincoli, ma utilizziamo questa volta l'ordine $d_2 = \{A, B, C, D, E, F\}$. Innanzitutto mostriamo l'ordine, poi troviamo il grafo indotto forzando le clique tra predecessori per ogni nodo a partire dal fondo e infine estraiamone il join tree:



Nuovamente, vediamo il risultato dei passaggi sopra: notiamo che abbiamo fatto poche aggiunte, che nella maggior parte delle volte significa che le clique sono rimaste di dimensione contenuta, e infatti hanno al massimo 3 variabili questa volta.



Infine mappo i vincoli alle clique:

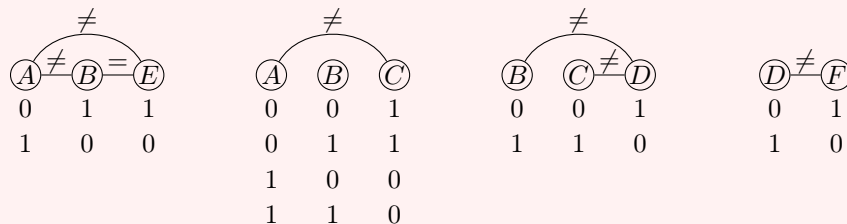
$\{A, B, C\}$ contiene R_{AC}
 $\{B, C, D\}$ contiene R_{BD}, R_{CD}
 $\{A, B, E\}$ contiene $R_{AE}, R_{BE}, \underline{R_{AB}}$ (*arbitrario*)
 $\{D, F\}$ contiene R_{DF}

Ogni vincolo è stato mappato ad ogni clique, quindi \mathcal{R} è una rete aciclica.

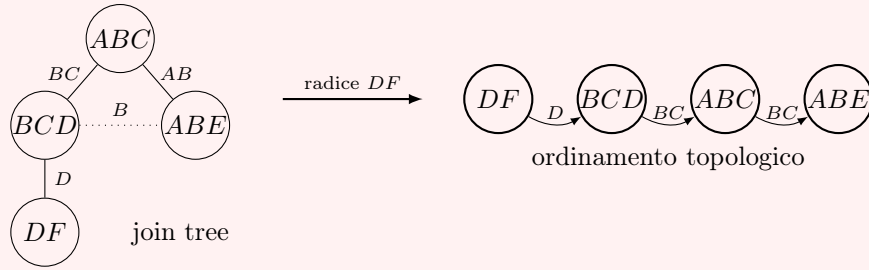
Ora vediamo come continua l'algoritmo: diamo le definizioni dei vincoli R_{uv} e dei loro domini, che sono

$$A \neq B, A \neq C, A \neq E, B = E, B \neq D, C \neq D, D \neq F \quad D_i = \{0, 1\}$$

e con questi vengono trovate le soluzioni di ogni clique; possiamo procedere per bruteforce, controllando la soddisfacibilità di ogni tupla possibile, oppure possiamo usare il backtracking, che per i casi reali è la scelta più efficiente. Nel nostro caso i domini e le clique hanno dimensioni ridotte quindi usiamo il bruteforce e scriviamo direttamente le tuple accettate:

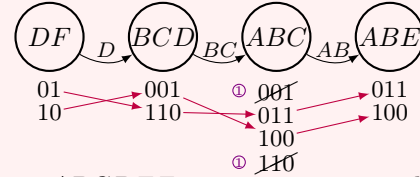


Ora semplicemente viene applicato TREE-SOLVER per trovare la soluzione: pongo i nodi del join tree in ordine topologico usando come radice arbitrariamente DF e poi faccio i REVISE necessari.



Come al solito, forziamo la consistenza diretta dalle foglie alle radici, mostrando ogni passo $R_k \leftarrow \pi_{S_k}(R_k \bowtie R_j)$

- ① $R'_{ABC} \leftarrow \pi_{ABC}(R_{ABC} \bowtie R_{ABE});$
- ② $R'_{BCD} \leftarrow \pi_{BCD}(R_{BCD} \bowtie R'_{ABC});$
- ③ $R'_{DF} \leftarrow \pi_{DF}(R_{DF} \bowtie R'_{BCD});$



Abbiamo due possibili soluzioni, $ABCDEF = 011011$ oppure $ABCDEF = 100100$, e prendiamo arbitrariamente la prima.

5 | Constraint Optimization

In molti problemi alcuni vincoli sono più blandi rispetto agli altri, in quando *opzionali* o *preferibili*: questi ci permettono di avere più libertà di scelta durante gli assegnamenti delle variabili, e quindi andiamo a formalizzarli per sfruttarli appieno. Estendiamo le reti a vincoli con i **soft constraint**, che esprimono una *preferenza* sull'assegnazione di una certa variabile, differentemente dagli **hard constraint**, che esprimono un *obbligo* sull'assegnazione (sono i vincoli che abbiamo usato finora).

Dato che ora abbiamo libertà di scelta, ci troviamo di fronte ad un problema di ottimizzazione, e quindi per trovare la soluzione dovremo non solo soddisfare ogni vincolo, ma anche minimizzare una *cost function* (o massimizzare una *objective function*) sulla base delle scelte possibili, ovvero i *soft constraint*. Tipicamente, avremo una funzione di costo/guadagno globale che è composta da una somma (per scheduling o planning) o un prodotto (per reti bayesiane o ragionamento su incertezze) di funzioni locali, che minimizzano/massimizzano un sottoinsieme di variabili.

Esempio

La manutenzione preventiva di un impianto di produzione di energia elettrica è un problema risolvibile con constraint optimization: la centrale non può smettere di produrre un minimo di energia utilizzando alcuni moduli, mentre altri possono essere schedulati in un certo ordine per minimizzare il costo di manutenzione.

Un altro problema sono le aste *combinatorie* online: dei compratori (*bidders*) fanno offerte su un *insieme* di oggetti, che vincono solo se non sono in conflitto con altre offerte contenenti oggetti in comune. In pratica, il battitore d'asta deve allocare i beni ai compratori in modo che gli oggetti in comune vengano dati ad una sola persona, ma deve anche massimizzare gli introiti della vendita.

5.1 Reti di costo

Formalmente ora, estendiamo le reti a vincoli con una funzione di costo: una **rete di costo** è definita come una quadrupla $\mathcal{C} = (X, D, C_h, C_s)$, dove

- $\mathcal{R} = \{X, D, C_h\}$ una rete a vincoli con hard constraint C_h ;
- $C_s = \{F_{Q_1} \dots F_{Q_\ell}\}$ un insieme di soft constraint con scope $Q_j \subseteq X$;

e una funzione di costo globale

$$F(X) = \sum_{j=1}^{\ell} F_j(X)$$

dove $F_j : (\times_{x_i \in Q_j} D_i) \rightarrow \mathbb{R}$ è una funzione di costo (o guadagno) locale sullo scope $Q_j \subseteq X$, che mappa le tuple ottenute dai domini delle variabili nello scope a valori reali. Tale funzione dovrà essere soddisfatta utilizzando assegnamenti a delle variabili X e notiamo che per definizione F_j utilizza solo le variabili nel suo scope, quindi $F_j(a) = F_j(a|_{Q_j})$. Inoltre, notiamo che due variabili sono connesse da un soft constraint se sono entrambe nello scope per una certa funzione F_j , quindi in pratica un soft constraint è una funzione di costo. Detto questo, il nostro scopo sarà trovare un assegnamento ottimale a^* che soddisfi \mathcal{R}_c e tale che $a^* = \arg \max_a F(a)$ oppure $a^* = \arg \min_a F(a)$.

Esempio

Un'asta combinatoria è espressa nel seguente modo:

- $S = \{s_1 \dots s_n\}$ insieme di oggetti all'asta;
- $B = \{b_1 \dots b_m\}$ insieme di offerte, in cui ogni offerta è nel formato $b_i = (S_i, r_i)$, dove $S_i \subseteq S$ ed r_i è il denaro offerto;

e l'obiettivo è di trovare un sottoinsieme di offerte $B' \subseteq B$ tale che nessuna offerta in B' condivide oggetti con un'altra e tale che la funzione di costo $C(B') = \sum_{b_i \in B'} r_i$ è massimizzata.

Cambiamo rappresentazione utilizzando una rete di costo, dove

- $X = \{b_i\}$ variabili di dominio $D_i = \{0, 1\}$, quindi offerte accettate o meno;

- $C_h = \{\forall i, j \exists R_{ij} \mid (b_i = 1, b_j = 1) \notin R_{ij}\}$, cioè ogni hard constraint rappresenta l'impossibilità di accettare due offerte che contengono gli stessi oggetti (questi vincoli vanno descritti a mano in base alla loro definizione vista sopra);
- $C_s = \{F_i(b_i) = r_i \cdot b_i\}$, cioè ogni soft constraint rappresenta il guadagno nel caso in cui una offerta venisse accettata;
- $F(B) = \sum_{b_i \in B} F_i(b_i)$ funzione di costo.

5.2 Tecniche di soluzione

Come per le reti a vincoli, possiamo risolvere una rete di costo tramite *ricerca* o *inferenza*, o ancora meglio con un mix di queste due. In generale avremo degli algoritmi specifici per gestire anche i soft constraint, in particolare: per la ricerca avremo approcci simili al backtracking, come *branch and bound* o *bucket elimination*, mentre per l'inferenza nuovamente forzeremo la consistenza, usando questa volta la *programmazione dinamica*.

Un'altra tecnica è di vedere la rete di costo come una serie di reti a vincoli, introducendo un nuovo hard constraint detto *cost bound*, il quale obbliga ogni funzione di costo F_j ad avere un costo minimo c^i . Si procede come segue:

1. data la rete di costo $\mathcal{C} = (X, D, C_h, C_s)$, introduciamo il cost bound c^i tra gli hard constraint, ottenendo una rete $\mathcal{R}^i = (X, D, C_h^i)$ dove $C_h^i = C_h \cup H^i$ e $H^i = F(a) \geq c^i$;
2. viene risolta la rete a vincoli iniziando c^i ad un valore *basso* che dipende dalle funzioni F_j (ad esempio se tutte le funzioni di costo sono strettamente positive possiamo impostare $c^1 = 0$) e poi aumentandone il valore sempre di più, fino ad arrivare a j , ovvero $c^1 \leq \dots \leq c^{k-1} \leq c^k$;
3. se arrivati alla soluzione a^k per il cost bound c^k non riusciamo a trovare una soluzione per c^{k+1} , allora la soluzione ottimale a^* è *bounded* in $Val(a^k) \leq Val(a^*) < c^{k+1}$, dove $Val(a) = \sum_{j=1}^{\ell} F_j(a)$.

Il vantaggio di questo approccio è che possiamo utilizzare tutte le tecniche efficienti che abbiamo visto per le reti a vincoli, ma lo svantaggio è che dobbiamo farlo su tante reti e non possiamo sapere a priori quante; ci concentreremo principalmente sulle tecniche sopra citate, che sono in generale molto più efficienti.

5.3 Branch and Bound

L'idea è di fare backtracking per trovare tutte le soluzioni, mantenendo man mano la migliore e scartando tutti i rami dell'albero che portano sicuramente ad una soluzione peggiore di quella corrente; se c'è un'inconsistenza, nessuna soluzione può essere trovata.

Assumiamo di voler *massimizzare* la funzione di costo: usiamo la miglior soluzione corrente come lower bound L e percorriamo l'albero delle soluzioni mediante tutte le tecniche viste finora per la ricerca negli alberi (Backtracking e Look-ahead), ignorando tutti i rami che portano a soluzioni di costo minore rispetto a L . In pratica, ogni volta che scendiamo in profondità nell'albero assegniamo un valore ad una variabile (come sempre), ma adesso utilizziamo inoltre una *bounding evaluation function* $f(\bar{a}_i)$ sull'assegnamento parziale \bar{a}_i costruito finora (fino alla variabile b_i), per controllare che il sottoalbero che andremo ad espandere dopo questo assegnamento abbia almeno una soluzione maggiore di L , in caso contrario possiamo semplicemente ignorare questo sottoalbero e cambiare l'assegnamento. Nel caso di una minimizzazione, impostiamo invece un upper bound U .

Impostare una buona *bounding evaluation function* è quindi una decisione cruciale per una ricerca efficiente, poiché deve essere

- *facile da computare*, altrimenti tanto vale fare una ricerca completa nell'albero;
- *più accurata possibile*, poiché se la funzione sovrastima troppo il costo non riusciamo a tagliare nessun ramo, ma se la funzione sottostima il costo tagliamo tutte le soluzioni migliori di quella corrente.

Un semplice approccio è la *First Choice bounding function*, che è simile alla tecnica del *forward checking* poiché ogni vincolo viene considerato separatamente:

$$f_{\text{FC}}(\bar{a}_i) = \sum_j \max_{a_{i+1} \dots a_n} F_j(\bar{a}_i, a_{i+1} \dots a_n)$$

In pratica dato un assegnamento parziale \bar{a}_i fino alla variabile i -esima, semplicemente calcola il costo massimo considerando ogni altra possibile variabile, senza controllare i vincoli tra loro, e quindi sovrastimando possibili perdite in caso di conflitti tra variabili. Più formalmente, dato che vogliamo massimizzare il costo effettivo $F(a)$, per definizione di $f_{FC}(a)$ non potremmo mai avere una sottostima, infatti in generale, per qualsiasi insieme di funzioni F_j , abbiamo

$$\max_a F(a) = \max_a \sum_j F_j(a) \leq \sum_j \max_a F_j(a) = f_{FC}(a) \implies \forall a, \max_a F(a) \leq f_{FC}(a)$$

Esempio

Dato il seguente problema di asta combinatoria

$$S = \{e_1 \dots e_7\} = \{TV, DVD, HomeTheatre, Decoder, StereoSystem, Beamer, NetBook\}$$

$$B = \{b_1 = (\{1, 2, 3, 4\}, 8), b_2 = (\{2, 3, 6\}, 6), b_3 = (\{1, 4, 5\}, 5), b_4 = (\{2, 7\}, 2), b_5 = (\{5, 6\}, 2)\}$$

estraiamone innanzitutto la rete di costo

$$X = \{b_1, b_2, b_3, b_4, b_5\} \text{ ognuna di domino } D_i = \{0, 1\}$$

$$C_h = \{R_{12}, R_{13}, R_{14}, R_{24}, R_{25}, R_{35}\}$$

$$C_s = \{F_1, F_2, F_3, F_4, F_5\} \text{ dove } F_i = b_i \cdot r_i$$

$$F(X) = \sum_{i=1}^5 F_i(X) = \sum_{i=1}^5 b_i \cdot r_i$$

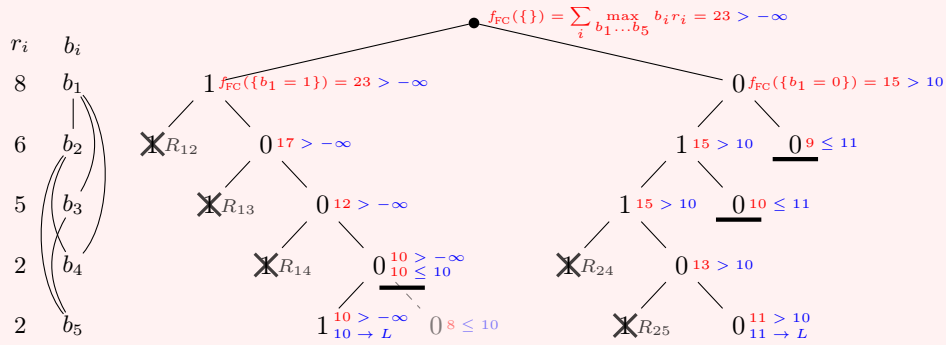
e poniamoci come obiettivo il massimizzare $F(X)$, quindi vogliamo trovare l'assegnamento

$$a^* = \max_X F(X) = \max_{b_1 \dots b_5} \sum_{i=1}^5 b_i \cdot r_i.$$

Espandiamo l'albero di ricerca usando l'ordine arbitrario $d = \{b_1, b_2, b_3, b_4, b_5\}$ e la *First Choice bounding function* per decidere se esplorare o meno un ramo, che ricordiamo essere

$$f_{FC}(\bar{a}_i) = \sum_j \max_{a_{i+1} \dots a_n} F_j(\bar{a}_i, a_{i+1} \dots a_n) = \sum_i \max_{b_i} b_i \cdot r_i$$

Mostriamo inoltre a sinistra i vincoli R_{uv} mediante il grafo primale, mentre sull'albero segniamo con $\times_{R_{uv}}$ l'impossibilità di assegnare un valore a causa del vincolo R_{uv} e a destra di ogni assegnamento il valore di $f_{FC}(\bar{a}_i)$ usando le variabili assegnate finora:



Commentiamo l'esecuzione: il lower bound iniziale è $L = -\infty$ e la discesa comincia arbitrariamente dai valori 1. Ad ogni passo calcoliamo f_{FC} e notiamo che è ovviamente ogni volta minore del lower bound corrente, $-\infty$, quindi possiamo proseguire con qualsiasi sottoalbero per ora ma, dopo aver impostato $b_1 = 1$, le variabili b_2, b_3, b_4 devono obbligatoriamente avere valore 0 a causa dei vincoli che le legano a b_1 , fino ad arrivare a $b_5 = 1$, che essendo la prima soluzione trovata imposta il nuovo valore del lower bound, $L = 10$. A questo punto torniamo indietro per cercare in $b_5 = 0$, ma notiamo che f_{FC} aveva dato 10, che ovviamente è il valore che già abbiamo, il che significa che per $b_5 = 0$ non avremo mai un sottoalbero che migliora la soluzione che già abbiamo, e quindi lo ignoriamo. Risaliamo l'albero poiché non c'è altro che possiamo fare e dato che f_{FC} per $b_1 = 0$ è maggiore di L scendiamo l'albero, continuando in questo modo fino a $b_5 = 0$, nel quale troviamo una nuova soluzione che migliora L . Risaliamo l'albero fino a $b_3 = 0$ e, dato che f_{FC} ha un valore minore di L , ignoriamo quel sottoalbero. Risaliamo ancora allora e ripetiamo per $b_2 = 0$, concludendo a ricerca con $L = 11$, quindi la soluzione migliore è rimasta $(0, 1, 1, 0, 0)$.

5.4 Bucket Elimination

La risoluzione di una rete di costo richiede la massimizzazione della funzione di costo $F(X)$, ma ogni soft constraint F_j ha uno scope che contiene solo *alcune* tra tutte le variabili. L'idea quindi è di scorrere tutte le variabili, massimizzare localmente tutti i soft constraint che le contengono e portare avanti i risultati ottenuti, continuando in questo modo: così facendo ad ogni nuova variabile che processiamo nell'ordine non dobbiamo preoccuparci di quelle precedenti, perché le abbiamo già massimizzate prima.

Più formalmente, dato un ordine arbitrario d di variabili, viene creato un insieme S contenente tutti i soft constraint F_j e, per ogni variabile x_i a partire dal fondo dell'ordine, l'algoritmo esegue una **backward phase** di due operazioni

1. BUCKETPARTITIONING: tutti i vincoli in S che contengono x_i nel loro scope vengono eliminati da S ed inserite nel **bucket** B_{x_i} . Se non ci sono vincoli che contengono la variabile di un certo bucket, quel bucket rimarrà vuoto;
2. BUCKETPROCESSING: si costruisce la funzione di costo *locale* H^{x_i} tramite l'operazione di **max marginalization** sui vincoli nel bucket B_{x_i}

$$H^{x_i}(t) = \max_{x_i} \sum_{f_k \in B_{x_i}} f_k(x_i, t) \quad \text{dove } t \text{ sono le altre variabili negli scope}$$

e la si inserisce in S . In pratica, ogni bucket B_{x_i} viene visto come una rete di costo a se stante, in cui i soft constraint f_k sono i vari vincoli F_j e H^{x_i} che sono stati assegnati al bucket, ovvero tutti e soli vincoli che dipendono dalla variabile x_i ; tale *sottorete* viene risolta come di consueto massimizzando la funzione di costo, che in questo caso è H^{x_i} , la quale rimane comunque da massimizzare rispetto a tutte le prossime variabili, e quindi deve essere inserita in S insieme a tutti gli altri vincoli.

Continuiamo così fino all'ultimo bucket, in cui $H^{x'}$ sarà un valore effettivo M , il quale ci permetterà di estrarre gli assegnamenti ottimali per le variabili e di ottenere la soluzione della rete. Per concludere quindi, a partire ora dall'inizio dell'ordine d , eseguiamo una **forward phase** di un'ultima operazione

3. VALUEPROPAGATION: viene calcolato l'assegnamento ottimale a_i^* per x_i tramite

$$a_i^* = \arg \max_{x_i} \sum_{f_k \in B_{x_i}} f_k(x_i, t)$$

e ad ogni calcolo successivo si rimpiazzano le variabili con l'assegnamento che abbiamo già calcolato.

In questo modo costruiremo un assegnamento parziale che massimizza la funzione di costo del bucket B_{x_i} a partire dal valore M e che verrà utilizzato per massimizzare tutte le funzioni per i bucket successivi, fino alla costruzione dell'assegnamento completo.

L'approccio è chiaramente quello della programmazione dinamica, in cui si costruisce la soluzione del problema in maniera incrementale a partire da quelle di sottoproblemi che lo compongono. Per le reti di costo questo approccio è molto conveniente perché riesce a sfruttare appieno la struttura del problema, risolvendolo localmente e poi propagando il minor numero di informazioni necessarie.

Esempio

Data la rete a vincoli di variabili $X = \{a, b, c, d, f, g\}$, con hard constraint che per ora ignoriamo e soft constraint

$$C_s = \{F_0(a), F_1(a, b), F_2(a, c), F_3(b, c, f), F_4(a, b, d), F_5(f, g)\}$$

l'obiettivo è trovare l'assegnamento α che massimizza la funzione di costo $F(X) = \sum_j F_j(X)$, cioè

$$\alpha = \max_{a, b, c, d, f, g} F_0(a) + F_1(a, b) + F_2(a, c) + F_3(b, c, f) + F_4(a, b, d) + F_5(f, g)$$

Notiamo che come detto prima intuitivamente, per massimizzare ad esempio d basta semplicemente massimizzare singolarmente il soft constraint F_4 . Applichiamo quindi l'algoritmo Bucket Elimination con l'ordine arbitrario $d_1 = \{a, c, b, f, d, g\}$.

$$S = \{\cancel{F_0^a}, \cancel{F_1^b}, \cancel{F_2^c}, \cancel{F_3^f}, \cancel{F_4^d}, \cancel{F_5^g}\} \cup \{H^g\} \cup \{H^d\} \cup \{H^f\} \cup \{H^b\} \cup \{H^c\} \cup \{H^a\}$$

BUCKET	VINCOLI DA S	MAX MARGINALIZATION
B_g	$F_5(f, g)$	$\rightarrow \max_g F_5(f, g) = H^g(f)$
B_d	$F_4(a, b, d)$	$\rightarrow \max_d F_4(a, b, d) = H^d(a, b)$
B_f	$F_3(b, c, f)$ $+ H^g(f)$	$\rightarrow \max_f F_3(b, c, f) + H^g(f) = H^f(b, c)$
B_b	$F_1(a, b)$ $+ H^d(a, b) + H^f(b, c)$	$\rightarrow \max_b F_1(a, b) + H^d(a, b) + H^f(b, c) = H^b(a, c)$
B_c	$F_2(a, c)$ $+ H^b(a, c)$	$\rightarrow \max_c F_2(a, c) + H^b(a, c) = H^c(a)$
B_a	$F_0(a)$ $+ H^c(a)$	$\rightarrow \max_a F_0(a) + H^c(a) = M$

Commentiamo l'esecuzione: innanzitutto inseriamo tutti i soft constraint F_j nell'insieme S e poi processando l'ordine al contrario eseguiamo le operazioni di BUCKETPARTITIONING e BUCKETPROCESSING ad ogni variabile. Viene creato il bucket per g , nel quale viene inserita solo $F_5(f, g)$, l'unica funzione che contiene g nel suo scope, e poi viene creata la funzione di costo locale $H^g(f)$, che massimizza g in tutte le funzioni in B_g , e che quindi avrà come scope le rimanenti variabili nelle funzioni (solamente f); infine, aggiungiamo H^g all'insieme S . La stessa cosa avviene per d , che riceverà solo la funzione $F_4(a, b, d)$ e genererà la funzione $H^d(a, b)$, nella quale d è già stata massimizzata, che inseriamo in S . Ora passiamo alla variabile f , la quale da S riceve sia $F_3(b, c, f)$, sia $H^g(f)$, poiché appunto sono tutte le funzioni che contengono f nel loro scope; massimizzando, otteniamo $H^f(b, c)$, che inseriamo in S , e così via fino alla variabile a , nella quale la funzione di costo locale $H^a()$ sarà semplicemente un intero M , che useremo nell'operazione di VALUEPROPAGATION, non trattata qui.

L'ordine delle variabili è quindi fondamentale e ne voglio uno che tenga i bucket più piccoli possibili, così da avere poche tuple durante la max marginalization: proprio perché dobbiamo fare tanti controlli quante sono le tuple per ogni bucket, Bucket Elimination sarà quindi esponenziale nella grandezza del bucket più grande.

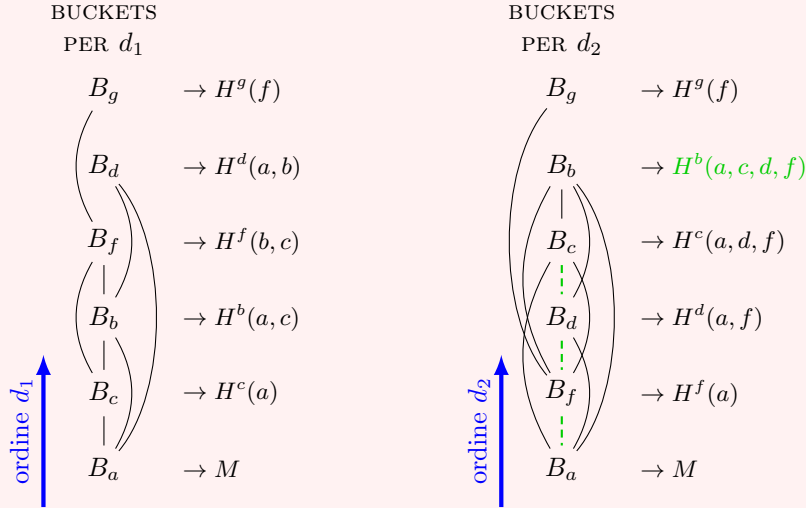
Esempio

Usando l'ordine $d_2 = \{a, f, d, c, b, g\}$ per la rete di prima notiamo che, dopo aver processato B_g contenente $F_5(f, g)$, il bucket B_b conterrà $F_1(a, b)$, $F_3(b, c, f)$, $F_4(a, b, d)$ e avremo una funzione di costo locale $H^b(a, c, d, f)$ che dipende da 4 variabili, e quindi avrà $|D_a| \cdot |D_c| \cdot |D_d| \cdot |D_f|$ tuple che poi dovremo controllare. Con questo ordine quindi già al secondo bucket abbiamo un numero considerevole di tuple da controllare, mentre con l'ordine precedente le funzioni $H^{x'}$ non avevano mai scope con più di 2 variabili e le tuple da controllare erano molte meno.

Notiamo immediatamente il collegamento molto forte con il Join Tree Clustering: per lo stesso ordine d delle variabili, la grandezza del bucket più grande coincide con l'ampiezza della clique più grande nel grafo indotto, poiché le variabili libere che compaiono in un bucket B_{x_i} sono esattamente i predecessori di x_i in base a quell'ordine, e quindi tutte le variabili nello scope di ogni $H^{x'}$ sono proprio i vincoli che forziamo nella rete, ovvero gli archi per le clique nel grafo indotto (in pratica, le funzioni $H^{x'}$ sono una codifica delle clique).

Esempio

Mettiamo a confronto i due ordini visti prima e osserviamo il collegamento con il grafo indotto: i soft constraint sono effettivamente vincoli tra le variabili, quindi li segniamo come al solito sull'ordine, inoltre mostriamo solamente le funzioni $H^{x'}$ di ogni bucket e i vincoli che aggiungono, tralasciando tutto il procedimento intermedio.



Notiamo che l'ordine d_1 è talmente buono da non aggiungere nemmeno un vincolo, e di conseguenza nessun arco, poiché le clique che considera sono talmente piccole da essere già presenti; per l'ordine d_2 invece dato che la variabile b , che ha molti vincoli, viene processata all'inizio, avremo una clique indotta di 5 vertici, che aggiunge 3 archi al grafo e quindi aumenta il numero di controlli da eseguire.

Per concludere, sappiamo che trovare l'ordine ottimale è un problema NP-completo, quindi ci dobbiamo affidare a delle euristiche, in particolare la MINWIDTH (che pone i nodi in ordine di maggior numero di vicini) o la MINFILL (che ragiona direttamente sul numero minimo di archi da aggiungere).

5.4.1 Trasformazione degli Hard Constraint

Per ora abbiamo considerato solo soft constraint, ovvero le *possibilità* che possiamo avere per le tuple, e ignorato gli hard constraint, ovvero gli *obblighi* che devono rispettare le tuple. Questo non è affatto un problema, poiché ogni hard constraint è riscrivibile come un soft constraint: dato un hard constraint R_s per un problema di massimizzazione (o minimizzazione), definiamo il suo soft constraint come

$$F_s(a) = \begin{cases} 0 & \text{se } a \text{ soddisfa } R_s \\ -\infty & \text{altrimenti (o } \infty) \end{cases}$$

In questo modo tutte le tuple che violano il vincolo hanno un valore talmente alto (o basso) che non verranno mai considerate, mentre le tuple valide valgono semplicemente 0, e quindi sono il lower bound (o upper bound) che non si potrà mai superare.

Esempio

In un problema di massimizzazione, per il vincolo $R_{uv} \equiv u \neq v$ le tuple accettate sono solamente $(0, 1)$ e $(1, 0)$, quindi costruiamo la funzione F_{uv} che per queste tuple ritorna 0, mentre per tutte le altre ritorna $-\infty$. In questo modo abbiamo trasformato un hard constraint R_{uv} in un soft constraint F_{uv} e possiamo utilizzare Bucket Elimination per risolvere il problema.

u	v	F_{uv}
0	0	$-\infty$
0	1	0
1	0	0
1	1	$-\infty$

Ora che consideriamo anche gli hard constraint nel Bucket Elimination, possiamo risolvere completamente una rete a vincoli: basta prima trasformare gli hard constraint in soft constraint e poi eseguire l'algoritmo.

5.4.2 Inclusione degli Hard Constraint

Non sempre trasformare gli hard constraint è la scelta più efficiente, infatti mantenendoli come tali possiamo ridurre di molto lo spazio di ricerca con gli algoritmi visti finora. Esiste quindi una versione di Bucket Elimination che mantiene gli hard constraint ed esegue una massimizzazione delle variabili solamente sulle tuple accettate: per ogni bucket B_{x_i} , generiamo la funzione di costo locale H^{x_i} massimizzando solo sulle tuple accettate, in pratica riscriviamo H^{x_i} come

$$H^{x_i}(t) = \max_{\{x_i | (x_i, t) \in R_{x_i}^{\times}\}} \sum_{f_k \in B_{x_i}} f_k(x_i, t)$$

dove $R_{x_i}^\times$ è il join tra tutti gli hard constraint che contengono la variabile x_i e quindi $(x_i, t) \in R_{x_i}^\times$ è una tupla accettata da tali hard constraint.

Esempio

Dato il problema dell'asta combinatoria visto prima,

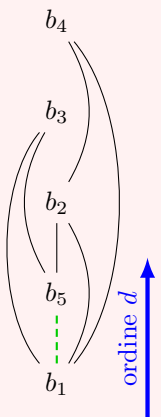
$$X = \{b_1, b_2, b_3, b_4, b_5\} \text{ ognuna di domino } D_i = \{0, 1\}$$

$$C_h = \{R_{12}, R_{13}, R_{14}, R_{24}, R_{25}, R_{35}\}$$

$$C_s = \{F_1, F_2, F_3, F_4, F_5\} \text{ dove } F_i = b_i \cdot r_i$$

$$F(X) = \sum_{i=1}^5 F_i(X) = \sum_{i=1}^5 b_i \cdot r_i$$

applichiamo la tecnica di inclusione degli hard constraint, che è più interessante ed efficiente (la tecnica di trasformazione si sviluppa esattamente come nell'esempio con i soli soft constraint, quindi l'abbiamo già vista). Applichiamo l'algoritmo Bucket Elimination con l'ordine arbitrario $d = \{b_1, b_5, b_2, b_3, b_4\}$, mostrando anche il grafo indotto, le assegnazioni degli hard constraint e l'offerta r_i di ogni variabile.

BUCKET PER	r_i	VINCOLI HARD	FUNZIONI CONTENUTE	MAX MARGINALIZATION
	2	$R_{14} \quad R_{24}$	$F_4(b_4)$	$\rightarrow \max_{\{b_4 (t, b_4) \in R_{14} \times R_{24}\}} F_4(t, b_4) = H^{b_4}(b_1, b_2)$
	5	$R_{13} \quad R_{35}$	$F_3(b_3)$	$\rightarrow \max_{\{b_3 (t, b_3) \in R_{13} \times R_{35}\}} F_3(t, b_3) = H^{b_3}(b_1, b_5)$
	6	$R_{12} \quad R_{25}$	$F_2(b_2)$ $+ H^{b_4}(b_1, b_2)$	$\rightarrow \max_{\{b_2 (t, b_2) \in R_{12} \times R_{25}\}} F_2(t, b_2) + H^{b_4}(b_1, b_2) = H^{b_2}(b_1, b_5)$
	2		$F_5(b_5)$ $+ H^{b_3}(b_1, b_5)$ $+ H^{b_2}(b_1, b_5)$	$\rightarrow \max_{b_5} F_5(b_5) + H^{b_3}(b_1, b_5) + H^{b_2}(b_1, b_5) = H^{b_5}(b_1)$
	8		$F_1(b_1)$ $+ H^{b_5}(b_1)$	$\rightarrow \max_{b_1} F_1(b_1) + H^{b_5}(b_1) = M$

Fino ad ora non abbiamo mai visto come calcolare effettivamente le H^{x_i} nella operazione di BUCKETPROCESSING, quindi ora vediamo una tecnica efficiente che ci permetterà anche di estrarre gli assegnamenti finali a_i^* senza aumentare la complessità spaziale.

Per creare ogni funzione H^{x_i} impostiamo una tabella suddivisa in due sezioni per quel bucket, una per gli hard constraint e una per le funzioni in B_{x_i} : partendo dagli hard constraint estraiamo le tuple ammesse, mappiamo la somma delle funzioni in B_{x_i} in base ad ogni tupla, ignoriamo la colonna della variabile x_i e infine estraiamo il massimo valore tra le tuple identiche. In questo modo costruiamo H^{x_i} ottimizzando x_i per fornire il miglior risultato possibile per ogni tupla ammessa, qualunque sia il valore di x_i .

b_1	b_2	b_4	$F_4(b_4)$	$H^{b_4}(b_1, b_2)$
0	0	0	0	$\max\{0, 2\} = 2$
0	0	1	2	
0	1	0	0	$\max\{0\} = 0$
1	0	0	0	$\max\{0\} = 0$
1	1	0	0	$\max\{0\} = 0$

b_1	b_5	b_3	$F_3(b_3)$	$H^{b_3}(b_1, b_5)$
0	0	0	0	$\max\{0, 5\} = 5$
0	0	1	5	
0	1	0	0	$\max\{0\} = 0$
1	0	0	0	$\max\{0\} = 0$
1	1	0	0	$\max\{0\} = 0$

b_1	b_5	b_2	$F_2(b_2)$	$H^{b_4}(b_1, b_2)$	$H^{b_2}(b_1, b_5)$
0	0	0	0	2	$\max\{2, 6\} = 6$
0	0	1	6	0	
0	1	0	0	2	$\max\{2\} = 2$
1	0	0	0	0	$\max\{0\} = 0$
1	1	0	0	0	$\max\{0\} = 0$

b_1	b_5	$F_5(b_5)$	$H^{b_3}(b_1, b_5)$	$H^{b_2}(b_1, b_5)$	$H^{b_5}(b_1)$
0	0	0	5	6	$\max\{11, 4\} = 11$
0	1	2	0	2	
1	0	0	0	0	$\max\{0, 2\} = 2$
1	1	2	0	0	

b_1	$F_1(b_1)$	$H^{b_5}(b_1)$	M
0	0	11	$\max\{11, 10\} = 11$
1	8	2	

Raggiunto il valore $M = 11$, ora applichiamo la fase di VALUEPROPAGATION: a partire dalla prima variabile x_i nell'ordine d , troviamo il valore che massimizza la funzione H^{x_i} (che sarà nel primo caso M), e seguiamo in questo modo fino a raggiungere l'ultima variabile, utilizzando via via i valori già ottenuti.

Nel nostro caso, a partire da b_1 cerchiamo il valore che dà $M = 11$, e visto che abbiamo già costruito le tabelle andiamo semplicemente a cercare il valore di b_1 nella riga che dà 11, cioè 0, quindi abbiamo che il valore ottimo è $b_1^* = 0$. A questo punto passiamo a b_5 , per la quale impostando $b_1 = 0$ abbiamo che H^{b_5} ha valore massimo 11 scegliendo $b_5^* = 0$; per b_2 impostando $b_1 = 0$ e $b_5 = 0$ abbiamo che H^{b_2} ha valore massimo 6 scegliendo $b_2^* = 1$, e così via, ottenendo infine $b_3^* = 1$ e $b_4^* = 0$. In conclusione, l'assegnamento ottimo è $a^* = (b_1^*, b_2^*, b_3^*, b_4^*, b_5^*) = (0, 1, 1, 0, 0)$.

6 | Incertezza

Abbiamo trattato nello specifico algoritmi utilizzati da agenti per risolvere specifici problemi: torniamo al cuore dell'intelligenza artificiale, gli agenti, e vediamo come gestire il raggiungimento della *migliore soluzione possibile* in un ambiente incerto.

Esempio

Pensiamo all'azione A_t di arrivare all'aeroporto in t minuti: ci sono un'infinità di variabili da tenere in considerazione, come errori nei sensori del traffico, incidenti, ruote bucate, ecc., quindi per evitare alcuni problemi è bene partire in anticipo, ad esempio con $t = 25$, e logicamente prima parto meglio è, ma impostare $t = 1440$ non è conveniente, dovrei dormire in aeroporto.

Per capire cosa è meglio fare introduciamo la **teoria della decisione**, la quale si sviluppa su due componenti fondamentali, la **teoria della probabilità**, che ci permette di considerare gli imprevisti (ruote bucate, ecc.), e la **teoria dell'utilità**, che ci permette di esprimere quello che preferiamo (evitare il pranzo in aeroporto, ecc.); con queste componenti andremo a scegliere l'azione che massimizza la **Maximum Expected Utility**, ovvero la maggiore utilità mediata su tutti i possibili risultati per quella azione. In generale, per gestire l'incertezza ci sono tre metodi, due dei quali molto simili:

default (o nonmonotonic) logic: si assume che un evento che è stato ignorato non avverrà mai;

fuzzy logic: un evento è sempre certo, ma si usano dei *gradi di verità* per darne la magnitudo;

probabilità: si descrive la possibilità che un evento avvenga in base all'evidenza disponibile dall'ambiente, cioè in base alle percezioni dell'agente.

Le asserzioni probabilistiche riassumono gli effetti di *pigrizia* (da parte del progettista) e di *ignoranza* (mancanza di evidenze), ponendoci davanti ad una incertezza dovuta a queste due cause. La probabilità verrà studiata nella sua accezione **soggettiva** (o **Bayesiana**), la quale studia le probabilità di un certo evento in base alle conoscenze che abbiamo al momento, e rappresenta quindi il nostro **grado di belief** rispetto al mondo.

6.1 Basi

Dato un certo esperimento, l'insieme Ω detto **sample space** contiene tutti i possibili risultati $\omega \in \Omega$ dell'esperimento, detti **sample point** (o **possible world** o **eventi atomici**). Diciamo **spazio probabilistico** (o **modello probabilistico**) un sample space in cui ogni $\omega \in \Omega$ ha una certa probabilità, cioè $0 \leq P(\omega) \leq 1$, e tale che $\sum_{\omega \in \Omega} P(\omega) = 1$, ovvero ad ogni esecuzione dell'esperimento si verifica un risultato ω . Chiamiamo **evento** un sottoinsieme $A \subseteq \Omega$ in cui più risultati sono ammessi, e quindi avremo $P(A) = \sum_{\omega \in A} P(\omega)$.

Esempio

Dato l'esperimento di lanciare un dado, il sample space $\Omega = \{1, 2, 3, 4, 5, 6\}$ rappresenta i possibili risultati dell'esperimento, ovvero ogni possibile faccia del dado. Notiamo che Ω è uno spazio probabilistico perché ogni risultato ha una probabilità di avvenire, in particolare $\forall \omega, P(\omega) = \frac{1}{6}$, e che la loro somma è 1. L'evento "tiro minore di 4" avrà quindi probabilità $P(\text{tiro} < 4) = P(1) + P(2) + P(3) = \frac{1}{2}$.

Possiamo pensare agli eventi come **proposizioni booleane**, le quali sono definite in funzione dei fattori presenti nell'ambiente: dato che tali fattori non sono sotto il nostro controllo, non possiamo sapere a priori il risultato che otterremo, e chiamiamo tali fattori **variabili casuali**. In base al caso useremo:

variabili casuali booleane con dominio è booleano. La variabile sarà indicata Var e per l'assegnazione $Var = true$ utilizzeremo la notazione var , o viceversa $\neg var$ per indicare $Var = false$;

variabili casuali discrete con dominio di valori mutualmente esclusivi (ad esempio $\langle sole, nuvoloso, pioggia \rangle$);

variabili casuali continue con dominio reale (ad esempio $Temperatura < 24.3$).

Un certo assegnamento di ogni variabile in un ambiente è quello che abbiamo chiamato **evento atomico**, ad esempio l'ambiente con $\{Temperatura, Meteo\}$ un evento atomico è $\{12.7, nuvolo\}$. Gli eventi atomici di un certo ambiente devono soddisfare le seguenti proprietà:

mutualmente esclusivi: può essere vero uno ed un solo assegnamento per volta;

esaustivi: ogni possibile caso deve essere coperto, quindi la loro disgiunzione deve essere sempre vera;

descrivono la verità di ogni proposizione: conoscere la probabilità di ogni evento atomico in una proposizione ci permette di calcolarne la probabilità;

generano ogni proposizione possibile: tramite disgiunzione degli eventi atomici rilevanti possiamo costruire qualsiasi proposizione, ad esempio $carie \equiv (carie \wedge dolore) \vee (carie \wedge \neg dolore)$;

Useremo la notazione $P(\omega)$ per indicare il **valore di probabilità** che l'evento ω ha di avvenire, mentre $\mathbf{P}(A)$ per indicare la **distribuzione di probabilità** che la variabile A ha per ogni suo valore, ovvero un vettore di valori di probabilità per ogni evento possibile su A .

Probabilità congiunta

La notazione $\mathbf{P}(A, B)$ oppure $\mathbf{P}(A \cap B)$ indica la **distribuzione congiunta** delle variabili casuali A e B , ovvero l'insieme delle probabilità di ogni combinazione di eventi possibili con A e B . Non necessariamente però le variabili devono essere legate tra loro, potremmo ad esempio chiedere $\mathbf{P}(HomeRun, Password)$.

Probabilità a priori

La notazione $P(\omega)$ indica la **probabilità a priori** di ω , ovvero la probabilità che, tra tutti gli eventi nell'intero campione dell'esperimento, si verifichi l'evento ω , ad esempio $P(Wheater = sunny) = 0.8$ significa che generalmente in un anno l'80% dei giorni è soleggiato. Per variabili casuali è un vettore.

Probabilità a posteriori

La notazione $P(\omega_1 | \omega_2)$ indica la **probabilità a posteriori** di ω_1 noto ω_2 , ovvero la probabilità che si verifichi l'evento ω_1 quando sappiamo ω_2 , ad esempio $P(Wheater = sunny | Temp = 34.5) = 0.95$ significa che sapendo che la temperatura attuale è di 34.5 gradi, la probabilità che questa sia una giornata soleggiata è del 95%. La definizione formale è

$$P(a | b) = \frac{P(a \wedge b)}{P(b)} \quad \text{e per variabili casuali} \quad \mathbf{P}(A | B) = \frac{\mathbf{P}(A, B)}{\mathbf{P}(B)}$$

Chain rule

In un ambiente con n variabili casuali, la distribuzione congiunta si può trovare applicando più volte la definizione di $\mathbf{P}(A|B)$

$$\mathbf{P}(X_1, \dots, X_n) = \prod_{i=1}^n \mathbf{P}(X_i | X_1, \dots, X_{i-1})$$

Esempio

$$\mathbf{P}(X_1, X_2, X_3) = \mathbf{P}(X_1, X_2) \mathbf{P}(X_3 | X_1, X_2) = \mathbf{P}(X_1) \mathbf{P}(X_2 | X_1) \mathbf{P}(X_3 | X_1, X_2)$$

quindi conoscendo solo $\mathbf{P}(X_1)$ possiamo evincere tutte le altre probabilità.

La domanda interessante ora è: ad un agente conviene davvero usare un modello probabilistico per descrivere l'ambiente? Nel 1931 de Finetti ha proposto la tesi "ragionare in termini delle leggi della probabilità è nell'interesse dell'agente", in quanto modellare l'agente in modo che il suo *belief* si basi sugli assiomi della probabilità, ovvero $P(\neg a) = 1 - P(a)$, $P(a \vee b) = P(a) + P(b) - P(a \wedge b)$ e $\sum_i P(\omega_i) = 1$, gli permette di avere un contatto maggiore con quella che è la realtà nella quale si trova.

6.1.1 Inferenza per enumerazione

Utilizziamo ora la logica proposizionale per descrivere gli eventi. Data una joint distribution, la probabilità di una qualsiasi proposizione ϕ è calcolata sommando gli eventi atomici ω per cui ϕ è vera:

$$P(\phi) = \sum_{\omega \models \phi} P(\omega).$$

Possiamo anche calcolare le probabilità condizionate, usando la definizione di $P(a|b)$ e i valori nella tabella.

Esempio

Nell'ambiente $\{Carie, Dolore, Sonda\}$ possiamo calcolare la probabilità $P(dolore)$ usando la distribuzione congiunta $\mathbf{P}(Carie, Dolore, Sonda)$

	dolore		$\neg dolore$	
	sonda	$\neg sonda$	sonda	$\neg sonda$
carie	0.108	0.012	0.072	0.008
$\neg carie$	0.016	0.064	0.144	0.576

semplicemente sommando tutti i valori di probabilità per gli eventi atomici che contengono *dolore*:

$$\begin{aligned} P(dolore) &= P(carie \wedge sonda) + P(carie \wedge \neg sonda) + P(\neg carie \wedge sonda) + P(\neg carie \wedge \neg sonda) \\ &= 0.108 + 0.012 + 0.016 + 0.064 = 0.2 \end{aligned}$$

Se voglio ora mostrare la relazione tra avere mal di denti e non avere una carie, calcolo la probabilità

$$P(\neg carie | dolore) = \frac{P(\neg carie \wedge dolore)}{P(dolore)} = \frac{0.016 + 0.064}{0.108 + 0.012 + 0.016 + 0.064} = 0.4$$

In un ambiente con variabili X , dati i valori e di alcune **variabili evidenza** $E \subset X$ che conosco, posso scoprire la distribuzione di probabilità di una **variabile query** $Y \in X$ rispetto ad essi computando ogni possibile caso dato dalle combinazioni dei valori delle **hidden variables** $H = X - Y - E$, di cui non so nulla:

$$\mathbf{P}(Y|E=e) = \alpha \mathbf{P}(Y, E=e) = \alpha \sum_h \mathbf{P}(Y, E=e, H=h)$$

dove α è il *fattore di normalizzazione* che vincola $0 \leq \mathbf{P}(Y|E=e) \leq 1$. Notiamo la somiglianza con la definizione di probabilità condizionata, in pratica $\alpha = \frac{1}{P(E=e)}$, ma ci torneremo tra poco.

Esempio

Noto il valore $Dolore = true$, se vogliamo calcolare la distribuzione di probabilità di *Carie* dobbiamo considerare tutte le variabili nascoste rimaste, ovvero *Sonda*, quindi

$$\begin{aligned} \mathbf{P}(Carie | dolore) &= \alpha \mathbf{P}(Carie, dolore) \\ &= \alpha [\mathbf{P}(Carie, dolore, sonda) + \mathbf{P}(Carie, dolore, \neg sonda)] \\ &= \alpha [\langle 0.108, 0.016 \rangle + \langle 0.012, 0.064 \rangle] \\ &= \alpha \langle 0.12, 0.08 \rangle \\ &= \langle 0.6, 0.4 \rangle \quad \text{con } \alpha = 5 \end{aligned}$$

Come abbiamo già detto, questa è una *distribuzione di probabilità*, ovvero il vettore dei valori di probabilità di *Carie* per ogni evento possibile sapendo *dolore*; il fattore α è semplicemente l'opposto della somma di tutti i valori nel vettore, nel nostro caso $\alpha = \frac{1}{0.12+0.08} = 5$, che è anche $\alpha = \frac{1}{P(dolore)} = \frac{1}{0.2} = 5$.

Il costo di questo approccio però è altissimo, infatti data d la dimensione del dominio più grande (nel nostro caso abbiamo tutte variabili booleane, quindi $d = 2$) e n il numero di variabili, il costo è $O(d^n)$ sia nel tempo (devo scorrere su tutti i valori delle variabili) che nello spazio (la tabella di distribuzione congiunta contiene $O(d^n)$ entry); un altro problema è l'avere la tabella in sé, in quanto non sempre l'agente conosce o può stimare tutte le probabilità degli eventi atomici.

Per ottenere la distribuzione congiunta quindi sfruttiamo gli **eventi indipendenti**, ovvero eventi tali che

$$\mathbf{P}(A|B) = \mathbf{P}(A) \quad \text{o anche} \quad \mathbf{P}(B|A) = \mathbf{P}(B) \quad \text{o anche} \quad \mathbf{P}(A, B) = \mathbf{P}(A)\mathbf{P}(B)$$

i quali ci daranno più tabelle separate, quindi complessivamente avremo meno entry nella tabella di interesse.

Esempio

Se vogliamo calcolare $\mathbf{P}(\text{Carie}, \text{Dolore}, \text{Sonda}, \text{Clima})$, dato che *Clima* è totalmente indipendente dalle altre tre variabili, possiamo separarlo in un'altra tabella a se stante di valori $\langle \text{sole}, \text{nuolo}, \text{pioggia} \rangle$, quindi

$$\mathbf{P}(\text{Carie}, \text{Dolore}, \text{Sonda}, \text{Clima}) = \mathbf{P}(\text{Carie}, \text{Dolore}, \text{Sonda}) \cdot \mathbf{P}(\text{Clima})$$

e il numero di entry complessive si riduce da $2^3 \cdot 3 = 24$ a $2^3 + 3 = 11$.

Spesso però è difficile anche dare una separazione netta tra le variabili, immaginiamo quante siano le possibili variabili che determinano la presenza o meno di una carie: possiamo considerare l'**indipendenza condizionale** tra variabili, ovvero tutti quei legami tra due o più variabili tali che, se ne vale una, tutte le altre sono ignorabili. Formalmente, se Y è condizionalmente indipendente da X noto E allora possiamo ignorare X , in formule

$$Y \perp\!\!\!\perp X \mid E \implies \mathbf{P}(Y \mid X, E) = \mathbf{P}(Y \mid E)$$

e grazie a tutte le indipendenze condizionali dell'ambiente, alla chain rule e all'assioma $P(a) = 1 - P(a)$ possiamo snellire la tabella al massimo, nella maggior parte dei casi passando da dimensione esponenziale in n a lineare.

Esempio

Se ho una carie, la probabilità che la sonda si incastri al suo interno non dipende in nessun modo dal fatto che io senta dolore, e ovviamente la stessa cosa vale se non sento dolore, quindi possiamo scrivere

$$\begin{aligned} P(\text{sonda} \mid \text{dolore}, \text{carie}) &= P(\text{sonda} \mid \text{carie}) \\ P(\text{sonda} \mid \text{dolore}, \neg \text{carie}) &= P(\text{sonda} \mid \neg \text{carie}) \end{aligned}$$

e in generale diciamo che *Sonda* è condizionalmente indipendente da *Dolore* quando *Carie* è nota, quindi

$$\begin{aligned} \text{Sonda} \perp\!\!\!\perp \text{Dolore} \mid \text{Carie} &\implies \mathbf{P}(\text{Sonda} \mid \text{Dolore}, \text{Carie}) = \mathbf{P}(\text{Sonda} \mid \text{Carie}) \\ \text{Sonda} \perp\!\!\!\perp \text{Dolore} \mid \text{Carie} &\implies \mathbf{P}(\text{Dolore} \mid \text{Sonda}, \text{Carie}) = \mathbf{P}(\text{Dolore} \mid \text{Carie}) \\ \text{Sonda} \perp\!\!\!\perp \text{Dolore} \mid \text{Carie} &\implies \mathbf{P}(\text{Dolore}, \text{Sonda} \mid \text{Carie}) = \mathbf{P}(\text{Dolore} \mid \text{Carie}) \cdot \mathbf{P}(\text{Sonda} \mid \text{Carie}) \end{aligned}$$

Ora possiamo finalmente calcolare l'intera distribuzione congiunta dell'ambiente:

$$\begin{aligned} \mathbf{P}(\text{Dolore}, \text{Sonda}, \text{Carie}) &= \mathbf{P}(\text{Dolore} \mid \text{Sonda}, \text{Carie}) \cdot \mathbf{P}(\text{Sonda}, \text{Carie}) && \text{chain rule} \\ &= \mathbf{P}(\text{Dolore} \mid \text{Sonda}, \text{Carie}) \cdot \mathbf{P}(\text{Sonda} \mid \text{Carie}) \cdot \mathbf{P}(\text{Carie}) && \text{chain rule} \\ &= \mathbf{P}(\text{Dolore} \mid \text{Carie}) \cdot \mathbf{P}(\text{Sonda} \mid \text{Carie}) \cdot \mathbf{P}(\text{Carie}) && \text{dato da } \perp\!\!\!\perp \end{aligned}$$

Dato che sono tutte variabili casuali hanno dominio booleano

- per $\mathbf{P}(\text{Carie})$ mi serve 1 solo parametro, infatti ho bisogno di sapere solo $P(\text{carie})$ poiché posso calcolare $P(\neg \text{carie}) = 1 - P(\text{carie})$, o viceversa;
- per $\mathbf{P}(\text{Dolore} \mid \text{Carie})$ ho 4 possibilità ma mi bastano 2 parametri
 - ★ con $P(\text{dolore} \mid \text{carie})$ posso ottenere, come prima, $P(\neg \text{dolore} \mid \text{carie})$, o viceversa;
 - ★ con $P(\text{dolore} \mid \neg \text{carie})$ posso ottenere $P(\neg \text{dolore} \mid \neg \text{carie})$, o viceversa;
- per $\mathbf{P}(\text{Sonda} \mid \text{Carie})$ come sopra, mi bastano 2 parametri.

Questo significa che con tutte queste semplici considerazioni abbiamo bisogno solamente di conoscere le probabilità di 5 eventi atomici, in particolare possiamo scegliere di voler stimare gli eventi $P(\text{carie})$, $P(\text{dolore} \mid \text{carie})$, $P(\text{dolore} \mid \neg \text{carie})$, $P(\text{sonda} \mid \text{carie})$ e $P(\text{sonda} \mid \neg \text{carie})$ per poi ottenere gli altri.

6.1.2 Regola di Bayes

Usando entrambe le definizioni $P(a \mid b)$ e $P(b \mid a)$ della probabilità condizionata otteniamo, come distribuzione

Teorema di Bayes

$$\mathbf{P}(A \mid B) = \frac{\mathbf{P}(B \mid A) \mathbf{P}(A)}{\mathbf{P}(B)} \quad \text{dove} \quad \mathbf{P}(B) = \sum_{a \in A} \mathbf{P}(B \mid a) P(a)$$

e quindi addiamo la capacità di scambiare ciò che ci è noto con quello che non conosciamo. Dato che stiamo lavorando con le distribuzioni possiamo riscrivere il denominatore come un **fattore di normalizzazione** definito in base alla distribuzione ottenuta, che vincola $0 \leq \mathbf{P}(A|B) \leq 1$, quindi

$$\mathbf{P}(Y|X) = \frac{\mathbf{P}(X|Y)\mathbf{P}(Y)}{\mathbf{P}(X)} = \alpha \mathbf{P}(X|Y)\mathbf{P}(Y)$$

Viene utile pensare in termini di cause ed effetti:

$$\mathbf{P}(Cause | Effetti) = \frac{\mathbf{P}(Effetti | Cause)\mathbf{P}(Cause)}{\mathbf{P}(Effetti)}$$

e notiamo la potenza di questa regola, che ci permette, in base alle nostre conoscenze a priori, di ricostruire le cause di un certo fenomeno osservandone solamente gli effetti.

6.1.3 Problema - Mondo del Wumpus

Poniamoci in un mondo disegnato a destra con il seguente PEAS:

Performance measure: prendere il Tesoro +1000, morire -1000, fare un passo -1;

Environment: un Eroe (l'agente) è in un mondo di $n \times n$ caselle, nelle quali sono presenti un Tesoro (l'obiettivo), il Wumpus (un mostro che mangia l'Eroe, solitamente si sposta ma in questo esempio lo immaginiamo immobile) e dei Pozzi. Il Tesoro emette un Luccichio nella casella in cui si trova, il Wumpus emana un forte Odore nelle caselle a lui adiacenti, mentre i Pozzi muovono una leggera Brezza nelle caselle a loro adiacenti;

Actions: andare dritto, ruotare a destra, ruotare a sinistra, raccogliere;

Sensors: occhi (per Luccichio), pelle (per Brezza) e naso (per Odore).

Per risolvere l'esempio intuitivamente, l'Eroe inizialmente non ha informazioni su nessuna casella, quindi può tentare la fortuna andando avanti in (1, 2), e percepire che c'è Odore, il che significa che in (1, 3), (2, 2) oppure in (1, 1) c'è il Wumpus; ovviamente scartiamo l'ipotesi (1, 1) poiché veniamo da lì e, dato che fare qualsiasi altra azione sarebbe rischioso e potrebbe portare al Wumpus, torniamo indietro a (1, 1) e poi muoviamoci verso destra. Ora, in (2, 1) percepiamo Brezza, il che significa che in (1, 1), (2, 2) oppure in (3, 1) c'è un Pozzo; ovviamente scartiamo l'ipotesi (1, 1), ma ora sappiamo qualcosa in più: se il Wumpus fosse in (2, 2) allora sentiremmo Odore anche qui, in (2, 1), ma non lo sentiamo, quindi il Wumpus è per forza in (1, 3), e la stessa cosa possiamo dire per il Pozzo, poiché se fosse in (2, 2) allora avremmo sentito Brezza anche in (1, 2), ma così non è stato, quindi il Pozzo è per forza in (3, 1), il che significa che (2, 2) è una casella safe e possiamo muoverci lì.

In un nuovo caso in cui sia (1, 2) che in (2, 1) ci fosse Brezza avremmo dei dubbi, poiché potrebbero esserci uno, due o tre Pozzi in (1, 3), (2, 2) e (3, 1), quindi abbiamo bisogno di fare dei ragionamenti in più, e qui entra in gioco la teoria della probabilità vista finora. Definiamo le variabili $P_{ij} = \text{true}$ se in (i, j) c'è un Pozzo e $B_{ij} = \text{true}$ se in (i, j) c'è Brezza: sappiamo che

$$b = \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1} \quad \text{known} = \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1}$$

e assumiamo che la presenza di un Pozzo sia indipendente dagli altri, cioè

$$\mathbf{P}(P_{1,1}, \dots, P_{4,4}) = \prod_{i,j=1,1}^{4,4} \mathbf{P}(P_{i,j}) = \underbrace{0.2^n}_{\text{arbitrario}} \times 0.8^{16-n} \text{ per } n \text{ Pozzi}$$

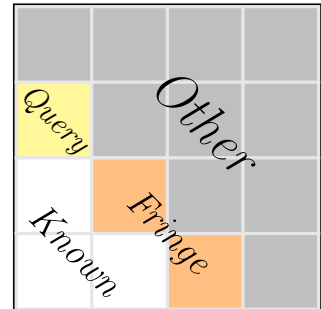
Ora, definito $Unknown = (\bigcup_{ij} P_{ij}) - P_{1,3} - Known$, possiamo eseguire la query $\mathbf{P}(P_{1,3} | \text{known}, b)$ scrivendo

$$\mathbf{P}(P_{1,3} | \text{known}, b) = \alpha \sum_{\text{unknown}} \mathbf{P}(P_{1,3}, \text{unknown}, \text{known}, b)$$

Definendo $Unknown = \text{Fringe} \cup \text{Other}$, notiamo che le osservazioni in $Known$ sono condizionalmente indipendenti da $Other$, quindi possiamo ottenere

$$\begin{aligned} \mathbf{P}(b | P_{1,3}, \text{Known}, \text{Unknown}) &= \mathbf{P}(b | P_{1,3}, \text{Known}, \text{Fringe}, \text{Other}) && \text{separazione di } Unknown \\ &= \mathbf{P}(b | P_{1,3}, \text{Known}, \text{Fringe}) && \text{indip. condiz. } \perp\!\!\!\perp \end{aligned}$$

4	O		B	P
3	W	OTB	P	B
2	O		B	
1	E	B	P	B
	1	2	3	4



e quindi vogliamo svolgere i calcoli della query in modo da ricondurci a quanto appena detto:

$$\begin{aligned}
\mathbf{P}(P_{1,3} \mid \text{known}, b) &= \alpha \sum_{\text{unknown}} \mathbf{P}(P_{1,3}, \text{known}, b, \text{unknown}) && \text{def. query} \\
&= \alpha \sum_{\text{unknown}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{unknown}) \mathbf{P}(P_{1,3}, \text{known}, \text{unknown}) && \text{chain rule} \\
&= \alpha \sum_{\text{fringe}} \sum_{\text{other}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}, \text{other}) \mathbf{P}(P_{1,3}, \text{known}, \text{fringe}, \text{other}) && \text{def. Unknown} \\
&= \alpha \sum_{\text{fringe}} \sum_{\text{other}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) \mathbf{P}(P_{1,3}, \text{known}, \text{fringe}, \text{other}) && \text{dato da } \perp\!\!\!\perp \\
&= \alpha \sum_{\text{fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) \sum_{\text{other}} \mathbf{P}(P_{1,3}, \text{known}, \text{fringe}, \text{other}) && \text{separazione} \\
&= \alpha \sum_{\text{fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) \sum_{\text{other}} \mathbf{P}(P_{1,3}) P(\text{known}) P(\text{fringe}) P(\text{other}) && \text{indip. di } P_{ij} \\
&= \alpha P(\text{known}) \mathbf{P}(P_{1,3}) \sum_{\text{fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) P(\text{fringe}) \underbrace{\sum_{\text{other}} P(\text{other})}_{=1 \text{ per def.}} && \text{separazione} \\
&= \alpha' \mathbf{P}(P_{1,3}) \sum_{\text{fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) P(\text{fringe}) && \text{semplificazione} \\
&= \alpha' \langle 0.2(0.04 + 0.16 + 0.16), 0.8(0.04 + 0.16) \rangle \approx \langle 0.31, 0.69 \rangle
\end{aligned}$$

Lo stesso identico calcolo si può fare anche per $P_{2,2}$ e $P_{3,1}$, ottenendo

$$\mathbf{P}(P_{2,2} \mid \text{known}, b) \approx \langle 0.86, 0.14 \rangle \quad \mathbf{P}(P_{3,1} \mid \text{known}, b) \approx \langle 0.31, 0.69 \rangle$$

e quindi l'agente rischierà meno decidendo di andare in (1, 3) oppure (3, 1) in quanto la probabilità di trovare un Pozzo in quelle caselle è del 31%, mentre per la casella (2, 2) è del 86%.

6.2 Sequential Decision Process

Finora abbiamo visto come prendere delle decisioni in un ambiente incerto per arrivare alla soluzione che massimizza l'utilità e minimizza il rischio dell'agente. Ora vediamo come agire "guardando al futuro", cercando quindi non di massimizzare l'outcome localmente, ma in previsione di ciò che più probabile che avvenga.

6.2.1 Sequenze di decisioni

Raramente nei casi reali si effettuano decisioni isolate, mentre invece è alla base del principio di causa/effetto prendere decisioni per raggiungere obiettivi a lungo termine e per acquisire informazioni dall'ambiente, quindi abbiamo bisogno di un formalismo per esprimere **sequenze di decisioni** in ambienti incerti.

Poniamoci ad esempio nel mondo qui a lato, dove identifichiamo

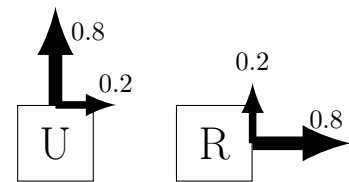
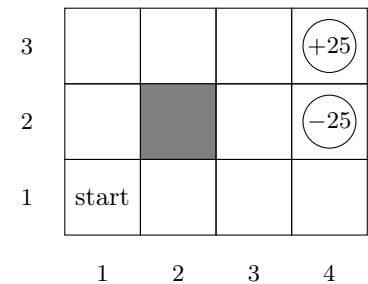
- gli stati $s \in S$ sono le caselle;
- le azioni $a \in A = \{R, U\}$ sono i 2 movimenti di spostamento in una casella adiacente, ma incerti: la probabilità di andare effettivamente nella direzione dell'azione è 0.8, mentre c'è una probabilità del 0.2 di compiere invece l'altra (ovviamente sbattere contro il muro o contro il limite del mondo riporta nella casella corrente);
- la **reward function**, che valuta le scelte dell'agente

$$R(s, a, s') = \begin{cases} -1 & \text{se } s' \text{ è non terminale} \\ \pm 25 & \text{se } s' \text{ è terminale} \end{cases}$$

- un **modello di transizione** $T(s, a, s')$, ovvero la capacità dell'agente di passare da s ad s' usando a . Per questo semplice esempio notiamo che $T(s, a, s') \equiv P(s' \mid s, a)$, ovvero il modello di transizione è la probabilità che fare l'azione a nello stato s mi porti ad s' .

L'agente vuole quindi trovare una strategia per raggiungere lo stato di goal +25 nel minor numero di passi possibile, massimizzando l'outcome totale.

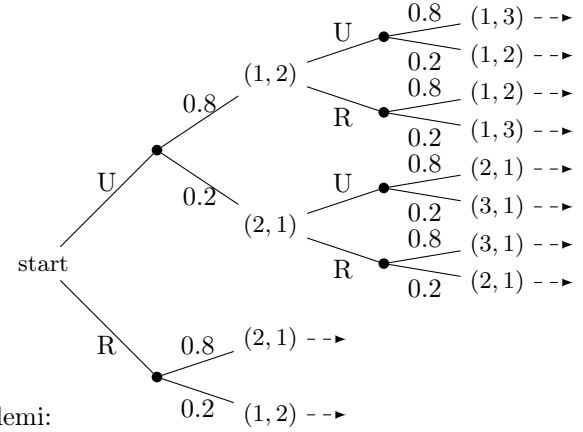
Nel nostro esempio, applicando la sequenza UURRR arriviamo allo stato finale con reward +25 nel caso in cui tutte le azioni si comportassero come desideriamo, ovvero con probabilità $0.8 \cdot 0.8 \cdot 0.8 \cdot 0.8 \cdot 0.8 = 0.32768$,



ma possiamo arrivarci anche quando le azioni si comportano diversamente: se dalla sequenza sopra otteniamo effettivamente le azioni [R][R][U][U]R raggiungiamo il +25 con probabilità $0.2 \cdot 0.2 \cdot 0.2 \cdot 0.2 \cdot 0.8 = 0.00128$, quindi in realtà con la sequenza UURRR la probabilità totale di raggiungere il goal +25 è $0.32768 + 0.00128 = 0.32896$. In generale quindi in un ambiente incerto dobbiamo considerare *tutte* le possibilità che possono portarci al goal desiderato, anche quelle in cui le cose non vanno come vorremmo ma che comunque ci portano al goal.

6.2.2 Transition Tree

Usiamo inizialmente il formalismo dei *transition tree*, in cui a partire dallo stato iniziale tracciamo degli archi, uno per ogni azione possibile, verso dei **chance nodes**, i quali si connettono agli stati successivi usando come etichetta dell'arco la probabilità di andare in quello stato a partire da quello corrente. L'albero può espandersi anche all'infinito (ad esempio nei casi in cui l'agente continua a sbattere contro il muro), ma dato che vogliamo fare un numero d finito di azioni, possiamo troncarlo alla profondità d : in questo modo, la probabilità di raggiungere il goal compiendo d azioni è il rapporto tra il numero di foglie goal e il numero di foglie totali.



Con questo formalismo abbiamo chiaramente dei problemi:

concettuale: valutare tutte le possibili azioni e agire semplicemente perché “con d azioni posso arrivare con probabilità $X\%$ allo stato goal”, senza considerare ciò che è effettivamente avviene dopo che un’azione viene compiuta nella realtà non aiuta a dare una strategia vincente per l’agente (devo tenere in conto di fare le azioni migliori in base a dove mi trovo);

pratico: la utility di una sequenza è più difficile da stimare rispetto alla utility dei singoli stati (non posso pensare che costruire “la strada più probabilisticamente buona” e seguirla ad occhi chiusi mi possa effettivamente *sempre* portare allo stato goal);

computazionale: abbiamo k azioni totali da poter compiere in una sequenza di d azioni massime, con n outcome possibili per azione, quindi ci sono $(kn)^d$ percorsi possibili da valutare nell’albero.

6.2.3 Policy

Cambiamo punto di vista: l’obiettivo dell’agente è trovare la sequenza di azioni che lo portano al goal con il miglior punteggio finale, in base alla reward function; vogliamo quindi una **policy ottimale** $\pi(s)$, cioè una funzione che, per ogni stato s , ritorna la migliore azione a da fare considerando i possibili outcome dovuti all’ambiente incerto in cui ci troviamo. Utilizziamo le policy perché descrivono intuitivamente il modo in cui ragioniamo: preferisco pagare penalità più alte ma seguire un percorso più sicuro, oppure voglio arrivare all’obiettivo rapidamente ma rischiare un po’ di più? In generale, diciamo che la policy ottimale massimizza la **expected sum of rewards**, ovvero decide quale azione fare in modo da ottenere il miglior punteggio *sperabile* in ogni scenario.

In base al modello di transizione e soprattutto alla reward function, la policy ottimale cambia: ad esempio, ponendo lo stato iniziale in basso a destra, se stare in uno stato non terminale avesse un costo altissimo, la policy cercherebbe di andare il più velocemente possibile verso uno stato terminale, anche il -25; se invece stare in un non terminale costasse pochissimo, allora potremmo permetterci di sbattere contro i muri pur di evitare con certezza di andare nel -25; ancora, se il costo fosse contenuto invece, la policy potrebbe decidere di promuovere il “percorso lungo”, pagando un po’ di più passando da sopra il muro per evitare il rischio di andare al -25, oppure optare per il “percorso breve”, pagando un po’ meno ma rischiando di andare nel -25.

Le policy possono essere costruite in modo da considerare più fattori oltre allo stato, infatti possono tener conto di “tempo rimasto” ed “elusione” di avversari. Possiamo categorizzare quindi le policy come:

non-stazionarie: $\pi : S \times T \rightarrow A$, quindi $\pi(s, t)$ ritorna la migliore azione da compiere nello stato s sapendo che dovrò fare ancora altre t azioni, cioè posso ancora cambiare al massimo t stati;

stazionarie: $\pi : S \rightarrow A$, quindi $\pi(s)$ ritorna la migliore azione da compiere nello stato s indipendentemente dalla progressione degli stati;

→	→	→	(+25)
↑		↑	(-25)
↑	←	←	←

policy ottimale per la $R(s, a, s')$
dell’ambiente visto prima
(incluso le azioni L e D)

stocastiche: $\pi(a | s)$ è la probabilità di scegliere l'azione a essendo nello stato s ; queste possono essere molto utili in ambienti concorrenti, ovvero nei quali ci sono più agenti che collaborano o che si scontrano tra loro (ad esempio, un malware, sapendo che possibilmente un antivirus lo sta osservando, ha interesse a compiere azioni evasive, quindi a volte farà azioni utili al suo attacco, altre volte azioni per confondere).

Da ora in avanti tratteremo solamente policy stazionarie, in quanto sono le più semplici da utilizzare, pur mantenendo un'alta espressività decisionale.

6.2.4 Utility

Come abbiamo già detto all'inizio, la teoria della decisione di basa su probabilità e utilità, quindi per costruire policy che rispettano le nostre preferenze dobbiamo formalizzare il concetto di utilità con le **utility function**. Innanzitutto, definiamo il concetto di *preferenza* come l'indicazione, tra tutte le sequenze di stati, di quelle che a noi sono interessanti (ad esempio, la preferenza di un percorso lungo si può esprimere come una qualsiasi sequenza di almeno k stati, oppure la preferenza di mantenere carica la batteria di un robot si può esprimere come una qualsiasi sequenza di stati raggiungibili con azioni a basso costo energetico).

Le preferenze saranno impostate arbitrariamente da noi e diremo che una sequenza di stati S è *preferibile* ad un'altra sequenza di stati S' utilizzando la notazione

$$S = [s_0, s_1, s_2, \dots] \succ [s'_0, s'_1, s'_2, \dots] = S'$$

Dato che lavoriamo con policy stazionarie, restringiamo il campo a **preferenze stazionarie**, e quindi possiamo supporre intuitivamente che

$$[s, s_0, s_1, s_2, \dots] \succ [s, s'_0, s'_1, s'_2, \dots] \iff [s_0, s_1, s_2, \dots] \succ [s'_0, s'_1, s'_2, \dots]$$

Sotto questa supposizione, è possibile dimostrare che le utility function sono costruibili solamente in 2 maniere:

utility function additiva: $U([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$;

utility function scontata: $U([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$;

dove $0 < \gamma \leq 1$ è il **fattore di sconto** della funzione, il quale diminuisce l'impatto dei reward man mano che proseguiamo con le scelte, rendendo le scelte iniziali più importanti rispetto a quelle successive. Notiamo che per come abbiamo definito le utility function, esse non sono altro se non la *expected sum of rewards* descritta in precedenza, e quindi possiamo ridefinire la policy ottimale come

Policy stazionaria ottimale

Dato un'ambiente di stati $s \in S$ in cui un agente può fare $a \in A$ azioni, in cui abbiamo definito una reward function $R(s, a, s')$ in base al modello di transizione $T(s, a, s') \equiv P(s' | s, a)$ che governa quell'ambiente, per cui abbiamo definito una utility function $U([\dots s_i \dots])$ per esprimere le nostre preferenze riguardo il comportamento dell'agente, la **policy ottimale** $\pi(s)$ per l'agente è la funzione che per ogni stato s ritorna l'azione a che porta nello stato s' e permette di massimizzare la utility function $U([s, s', \dots])$.

Efficacia Come valutiamo l'efficacia di una policy? Definiamo la **value function** $V : S \rightarrow \mathbb{R}$, che associa il valore $v_\pi(s)$ ad ogni stato s considerando gli expected accumulated reward da lì in poi utilizzando la policy. Ad esempio, per la policy ottimale vista sopra, il valore $v_\pi(S_{2,3})$ sarà la somma dei reward ottenuti seguendo la policy $\pi(s)$ a partire da $S_{2,3}$, ovvero spostandosi prima a destra per andare in $S_{3,3}$ e di nuovo a destra per raggiungere il goal +25, quindi $v_\pi(S_{2,3}) = -1 + 25 = 24$.

Sequenze infinite Come possiamo conoscere il valore $v_\pi(s)$ se le sequenze di azioni sono infinite? Non riusciremmo mai a smettere di sommare i reward accumulati. Abbiamo 3 principali soluzioni:

orizzonte finito: scegliere un numero T di passi dopo il quale la sequenza termina forzatamente, ovvero limitare le sequenze a T stati al massimo, ma questo produce policy non-stazionarie, quindi lo ignoriamo;

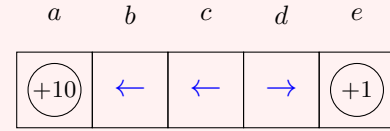
assorbire stati: garantire che per ogni possibile policy si raggiungerà sempre uno stato terminale, ma questa è un'assunzione troppo pesante da fare senza perdere generalità, quindi la ignoriamo;

usare uno sconto: nell'ipotesi che tutti i reward siano limitati, utilizziamo una utility function scontata con $0 < \gamma < 1$, in quanto la utility sarà limitata e convergente

$$U([s_0, \dots, s_\infty]) = \sum_{t=0}^{\infty} \gamma^t R(s_t) \leq \frac{R_{max}}{1 - \gamma}$$

Esempio

Posti nel mondo a lato, possiamo muoverci solamente a destra o a sinistra, senza incertezza, e gli unici reward che abbiamo sono gli stati agli estremi del “corridoio” (tutti gli altri stati valgono 0). Costruiamo la policy ottimale con fattore di sconto $\gamma = 0.1$, cioè massimizziamo la utility function degli stati b , c e d (ignoriamo il tornare indietro poiché peggiore in ogni caso):



- per b andando subito a sinistra otteniamo 1 mentre se andiamo sempre a destra otteniamo 0.001

$$U([b, a]) = \gamma^0 \cdot 0 + \gamma^1 \cdot 10 = 1 \quad U([b, c, d, e]) = \gamma^0 \cdot 0 + \gamma^1 \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 1 = 0.001$$

quindi ovviamente scegliamo di compiere ←;

- per c andando sempre a sinistra otteniamo 0.1 mentre se andiamo sempre a destra otteniamo 0.01

$$U([c, b, a]) = \gamma^0 \cdot 0 + \gamma^1 \cdot 0 + \gamma^2 \cdot 10 = 0.1 \quad U([c, d, e]) = \gamma^0 \cdot 0 + \gamma^1 \cdot 0 + \gamma^2 \cdot 1 = 0.01$$

quindi ovviamente scegliamo di compiere ←;

- per d andando subito a sinistra otteniamo 0.01 mentre se andiamo sempre a destra otteniamo 0.1

$$U([d, c, b, a]) = \gamma^0 \cdot 0 + \gamma^1 \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 10 = 0.01 \quad U([d, e]) = \gamma^0 \cdot 0 + \gamma^1 \cdot 1 = 0.1$$

quindi ovviamente scegliamo di compiere →.

6.2.5 Alberi di decisione

Introduciamo il concetto di **decision tree**, che ci permette di definire in maniera formale la policy ottimale, utilizzando il processo di **backward induction** (o **rollback**, in pratica è l'algoritmo *expectimax*): viene costruito il transition tree come visto prima, ma ora le foglie sono il reward degli stati goal e gli stati diventano **decision nodes**; in questo modo possiamo tornare indietro e computare la *expected utility* del chance node precedente ad ogni foglia, per poi tornare ancora indietro e computare quella massima tra tutti i chance node per questo decision node, e ripetere fino ad arrivare alla radice.

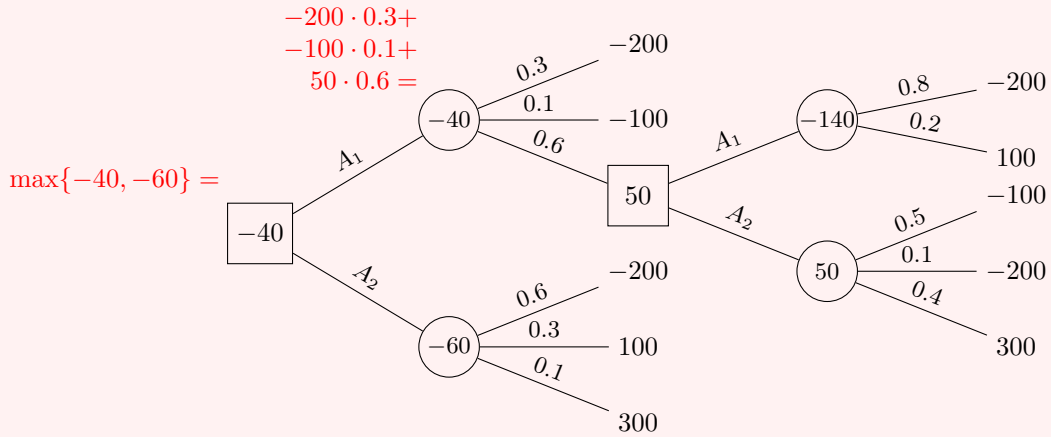
Formalmente quindi, costruito un decision tree, computiamo la *Maximum Expected Utility* a partire dalle foglie, utilizzando le funzioni di *expected utility* $EU(\cdot)$ per ogni elemento che lo compone:

- ogni nodo foglia G ha $EU(G) = V(G)$, ovvero il valore di quello stato goal;
- ogni chance node C ha $EU(C) = \sum_{D \in \text{Child}(C)} P(D) EU(D)$, dove $P(D)$ è la probabilità di andare in D ;
- ogni decision node D ha $EU(D) = \max_{C \in \text{Child}(D)} EU(C)$, dove C sono i chance node di D .

Abbiamo detto che, per un certo stato s , la policy ritorna l'azione migliore da fare in previsione di ciò che succederà, e utilizzando i decision tree questo risulta immediato: semplicemente ritorniamo l'azione che massimizza la MEU per il decision node D_s associato ad s , ovvero

$$\pi(D_s) = \arg \max_{C \in \text{Child}(D_s)} EU(C)$$

Esempio



6.3 Markov Decision Process

I decision tree sono ottimi per il calcolo della MEU, ma sono alberi appunto, e quindi non sono ottimizzati nello spazio delle decisioni. Utilizziamo quindi i **Markov Decision Processes** o **MDP**, che sono una versione estremamente più compatta (a grafo) dei decision tree, definiti come una quadrupla $\langle S, A, R, T \rangle$ dove

- S è un insieme finito di n stati
- A è un insieme finito di m azioni;
- $T(s, a, s')$ è la transition function, cioè $P(S_{t+1} = s' | S_t = s, A_t = a)$, dove t indica la progressione;
- $R(s, a, s')$ è la reward function, cioè $\mathbb{E}[R_{t+1} | S_{t+1} = s', A_t = a, S_t = s]$, dove abbiamo \mathbb{E} poiché anche il reward può essere incerto;

Studiamo i MDP perché adottano una filosofia interessante: dato uno stato presente, il futuro è indipendente dal passato; in pratica le azioni e gli stati passati sono irrilevanti per le decisioni future, e quindi possiamo ottenere delle policy che permettono di creare *simple reflex agents* che massimizzano la expected utility. Gli MDP infatti rispettano 3 proprietà chiave per quanto riguarda la creazione di policy:

Dimaniche di Markov: $P(R_{t+1}, S_{t+1} | S_0, A_0, R_1 \dots R_t, S_t, A_t) = P(R_{t+1}, S_{t+1} | S_t, A_t)$;

Stazionarietà: $\forall t, t', P(R_{t+1}, S_{t+1} | S_t, A_t) = P(R_{t'+1}, S_{t'+1} | S_{t'}, A_{t'})$;

Osservabilità totale: non possiamo predire esattamente il prossimo stato ma sappiamo sempre dove siamo, quindi una volta cambiato stato possiamo sapere quale esso sia. Quest'ultimo punto è un'assunzione molto pesante, in quanto nella maggior parte dei casi reali abbiamo osservabilità parziale, quindi non sappiamo con certezza nemmeno dove ci troviamo.

Concretamente, un MDP è rappresentato da un **transition graph** (quindi utilizzano la notazione snella dei transition tree e la logica robusta dei decision tree), in cui ogni arco che connette chance node a stati contiene anche la reward (associata come al solito al passaggio dallo stato corrente a quello successivo con quell'azione).

Esempio

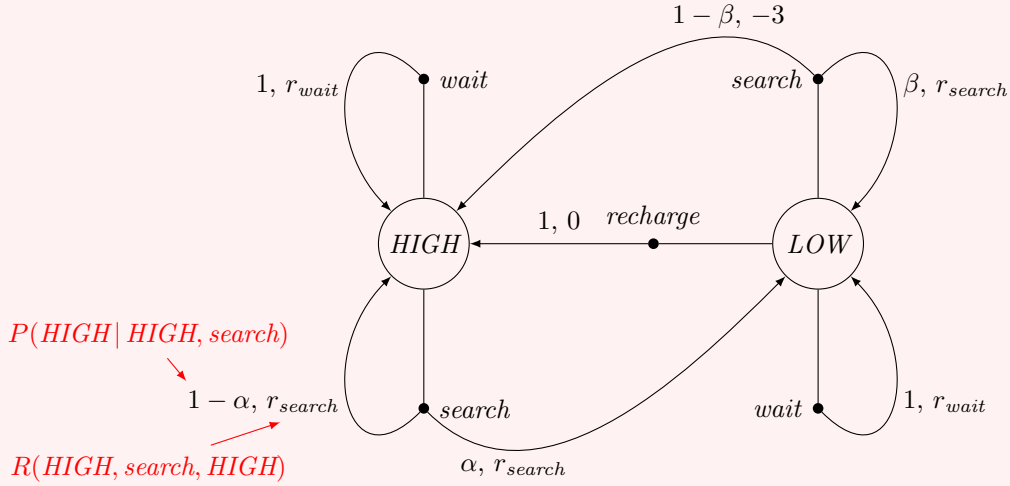
Un robot autonomo ricicla-lattine compie azioni differenti in base al livello di carica della sua batteria: se è alto allora può attivamente cercare le lattine, abbassando la sua carica con probabilità α , mentre se è basso allora può comunque cercare attivamente lattine rimanendo carico con probabilità β , oppure tornare a ricaricarsi senza incertezza alla sua docking station. In entrambi gli stati di batteria, il robot può ottenere passivamente delle lattine stando fermo, nel caso in cui qualcuno glielo porti. Nel caso in cui avendo poca batteria, a seguito di una ricerca attiva, il robot dovesse scaricarsi, allora un operatore umano potrebbe riportarlo manualmente alla docking station per ricaricarsi, ma subirebbe una punizione di -3 per averlo fatto "scomodare".

Notiamo che in questo caso l'insieme delle azioni varia in base allo stato in cui si trova l'agente:

$$A(HIGH) = \{search, wait\}$$

$$A(LOW) = \{search, wait, recharge\}$$

è una dinamica che non abbiamo mai affrontato, ma la modellazione utilizzando i MDP risulta immediata. Per questo robot avremo il seguente transition graph, nel quale i vari reward sono stati lasciati incogniti poiché non propriamente specificati.



6.3.1 Risolvere un MDP

Con le nozioni di policy, utility function e valore, definiamo per un MDP

value function: $v_\pi(s) = \mathbb{E} [\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a]$, ovvero il valore dello stato s quando si segue la policy $\pi(s)$ che usa una utility function scontata;

Q-value function (o quality function): $q_\pi(s, a) = \sum_{s'} T(s, a, s') (R(s, a, s') + \gamma v_\pi(s'))$, ovvero il valore del fare l'azione a nello stato s seguendo la policy $\pi(s)$.

Notiamo che concettualmente $v_\pi(s) = q_\pi(s, \pi(s))$, quindi il valore di uno stato è definibile ricorsivamente utilizzando i valori dei suoi stati successori:

Equazione di Bellman

Data una policy $\pi(s)$, l'equazione

$$v_\pi(s) = \sum_{s'} p(s' | s, \pi(s)) (R(s, \pi(s), s') + \gamma v_\pi(s'))$$

è una condizione di self-consistency per la policy, ovvero la garanzia che una volta cominciata a seguire la policy, il valore delle sue decisioni sarà consistente nel tempo poiché basato sui valori delle scelte future.

Essendo ricorsiva, abbiamo degli evidenti problemi a calcolare questa equazione, ma per ora non ce ne preoccupiamo e continuiamo con lo studio teorico. Sfruttiamo questa equazione per ottenere la policy che a noi davvero interessa, ovvero quella ottima:

- $\pi_*(s)$ è una policy ottima se e solo se $\forall s, \pi, v_*(s) \geq v_\pi(s)$;
- $v_*(s) = \max_\pi v_\pi(s)$ è il valore ottimale tra quelli delle possibili policy;
- $q_*(s) = \max_\pi q_\pi(s, a)$ è la quality function ottimale tra quelle delle possibili policy;

Dato che anche la policy ottima deve soddisfare la condizione di self-consistency, anche per essa deve valere l'equazione di Bellman, ma possiamo effettuare una riscrittura:

Equazione di Bellman per l'ottimalità

Data la policy ottimale $\pi_*(s)$, l'equazione

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s, a) = \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v_*(s'))$$

ne descrive la self-consistency, dove $\mathcal{A}(s)$ è l'insieme delle azioni che si possono compiere nello stato s .

Ora possiamo finalmente andare a definire gli algoritmi calcolano le policy.

6.3.1.1 Policy Iteration

Un approccio alla costruzione delle policy è l'algoritmo **Policy Iteration**, proposto da Howard nel 1960: dato che per calcolare la policy ottimale mi serve la value function ottimale e viceversa, calcoliamo entrambe iterativamente a partire da un'assegnazione arbitraria, aggiornando la policy corrente con la value function corrente (**policy evaluation**) e viceversa (**policy improvement**). In pratica abbiamo

$$\pi_0 \xrightarrow{\text{Eval}} v_{\pi_0} \xrightarrow{\text{Impr}} \pi_1 \xrightarrow{\text{Eval}} v_{\pi_1} \xrightarrow{\text{Impr}} \pi_2 \xrightarrow{\text{Eval}} \dots \xrightarrow{\text{Eval}} v_* \xrightarrow{\text{Impr}} \pi_*$$

Nella fase di *policy evaluation*, in cui calcoliamo la value function, possiamo utilizzare

- l'equazione di Bellman con la policy corrente, risolvendo un sistema di $|S|$ equazioni lineari di $|S|$ incognite (poiché ogni stato ha un valore che dipende dai valori degli altri stati), quindi impiegando $O(|S|^3)$ con algoritmi di algebra lineare (abbastanza costoso); oppure
- un'approssimazione iterativa, in cui semplicemente valutiamo man mano tutti i valori della value function utilizzando l'equazione di Bellman con la policy corrente e raffiniamo l'errore Δ di questa valutazione iterata fino a una soglia ε , impiegando $O(|S|^2)$ ad ogni iterazione (più efficiente);

mentre nella fase di *policy improvement*, in cui calcoliamo la policy, dato che ora ho il nuovo valore di tutti gli stati, cambio tutti le azioni della policy in base ad essi; nel caso in cui ci fosse ancora margine di miglioramento (ovvero se le azioni della policy sono state effettivamente cambiate) allora ricominciamo dalla policy evaluation, se invece dall'ultima iterazione la policy non è cambiata allora abbiamo concluso.

Algorithm 17 Policy Iteration

```

1: inizializza gli array  $v[]$  e  $\pi[]$  arbitrariamente // ad esempio tutto a 0
2:
3: repeat
4:   // Policy Evaluation
5:   repeat
6:      $\Delta \leftarrow 0$  // errore della stima
7:     for all  $s \in S$  do
8:        $old\_value \leftarrow v[s]$ 
9:        $v[s] \leftarrow \sum_{s'} p(s' | s, \pi[s]) (R(s, \pi[s], s') + \gamma v[s'])$  // equazione di Bellman
10:       $\Delta \leftarrow \max(\Delta, |old\_value - v[s]|)$ 
11:   until  $\Delta < \varepsilon$  // dove  $\varepsilon$  è un intero positivo piccolo, l'accuratezza
12:
13:   // Policy Improvement
14:    $policy\_stable \leftarrow true$ 
15:   for all  $s \in S$  do
16:      $old\_action \leftarrow \pi[s]$ 
17:      $\pi[s] \leftarrow \arg \max_a \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v[s'])$  // miglioriamo la policy
18:     if  $old\_action \neq \pi[s]$  then
19:        $policy\_stable \leftarrow false$  // c'è ancora margine di miglioramento
20:
21: until not  $policy\_stable$ 
22: // nessun ulteriore miglioramento, terminiamo
23: return  $\pi, v$ 

```

In questo modo abbiamo la garanzia che la policy verrà migliorata ad ogni iterazione, convergendo molto più velocemente rispetto a Value Iteration, il prossimo algoritmo, ma con un costo computazionale molto maggiore per ogni iterazione (devo ogni volta trovare $\arg \max_a$ per ogni stato, ad ogni iterazione ...).

6.3.1.2 Value Iteration

L'idea alla base è di incorporare policy evaluation e improvement della Policy Iteration, utilizzando l'equazione di Bellman per l'ottimalità come una *regola di update* per ottenere la value function iterativamente, e con questa costruire la policy in maniera greedy: chiamiamo il metodo **Value Iteration** e, come il precedente, è una approssimazione iterata, implementabile tramite programmazione dinamica. Innanzitutto definiamo

Bellman back-up function

Esprimendo l'equazione di Bellman per l'ottimalità in relazione all'iterazione k otteniamo

$$v_{k+1}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v_k(s'))$$

in questo modo possiamo calcolare il valore di ogni stato utilizzando i “back-up” dei valori precedenti, quindi la soluzione del passo $k + 1$ dipende dal passo k , e possiamo scrivere l'algoritmo iterativo di approssimazione

Algorithm 18 Value Iteration

```

1: inizializza l'array  $v[]$  arbitrariamente // ad esempio tutto a 0
2: repeat
3:    $\Delta \leftarrow 0$ 
4:   for all  $s \in S$  do
5:      $old\_value \leftarrow v[s]$ 
6:      $v[s] \leftarrow \max_a \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v[s'])$  // Bellman back-up function
7:      $\Delta \leftarrow \max(\Delta, |old\_value - v[s]|)$ 
8: until  $\Delta < \varepsilon$ 
9: for all  $s \in S$  do
10:   $\pi[s] \leftarrow \arg \max_a \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v[s'])$ 
11: return  $\pi$ 

```

Value Iteration garantisce di convergere verso la value function ottimale, quindi con ε infinitamente piccolo produrrà sempre la policy stazionaria ottimale, e quindi darà un risultato efficiente da tenere in memoria (in pratica è un array di azioni, dove ogni indice rappresenta uno stato). La complessità di Value Iteration è $O(|S|^2 \cdot |A|)$, quindi risulta particolarmente prestante per ambienti con pochi stati e molte azioni.

6.4 Osservabilità parziale

Nei casi reali ci troviamo di fronte ad ambienti parzialmente osservabili, nei quali l'agente non sa con certezza in quale stato si trova e quindi non può utilizzare le policy finora descritte. Definiamo un **Partially Observable Markov Decision Process** (o **POMDP**) una quintupla $\langle S, A, R, T, O \rangle$, dove l'aggiunta $O(s, e) \equiv P(e | s)$ è il **modello di osservazione** che definisce la probabilità che l'agente ottenga l'evidenza e nello stato s .

Teorema (Astrom, 1965)

La policy ottimale di un POMDP è una funzione $\pi(b)$ dove b è il **belief state** dell'agente, ovvero una distribuzione di probabilità sugli stati.

Possiamo quindi trasformare un POMDP in un MDP nello spazio dei belief state, dove avremo $T(b, a, b')$. Le policy dei POMDP saranno sempre incentrate sul raccogliere il maggior numero di informazioni possibili durante il raggiungimento dello stato goal, così da potersi muovere con più certezza sulla propria posizione (ad esempio, un agente che deve uscire da una stanza “ad occhi chiusi” e si trova vicino ad un muro sicuramente seguirà il muro, poiché tale informazioni gli dà una certezza in più sulla sua posizione). Sfortunatamente, risolvere un POMDP è un problema PSPACE-hard, e quindi pressoché intrattabile senza fare delle assunzioni, le quali ci permettono di rappresentare con un certo margine di bontà un ambiente incerto come una MDP e quindi risolvere il problema sufficientemente bene.

Esempio

Ponendoci nuovamente nell'ambiente precedente, in cui ricordiamo che il reward $R(s, a, s') = -1$ per ogni stato, costruiamo la policy ottimale per lo stato x usando Value Iteration quando $\gamma = 0.5$ (mostriamo solo due iterazioni). Innanzitutto impostiamo come valore iniziale degli stati $\forall s \setminus \{+25, -25\}$, $v_0(s) = 0$ mentre per gli stati goal abbiamo $v_0(+25) = 25$, $v_0(-25) = -25$.

Ricordando che la Bellman back-up function è

$$v_{k+1}(s) = \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v_k(s'))$$

3			x	$\oplus 25$
2				$\ominus 25$
1				
	1	2	3	4

vediamo come avviene il calcolo della prima iterazione per x :

$$v_1(x) = \max_{a \in \mathcal{A}(x)} \begin{cases} \underbrace{0.8 \cdot (-1 + \gamma \cdot 0)}_{v_0(x)} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 0)}_{v_0(S_{2,3})} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 25)}_{v_0(+25)} = 0.25 & \text{con l'azione U} \in \mathcal{A}(x) \\ \underbrace{0.8 \cdot (-1 + \gamma \cdot 25)}_{\text{goes in } +25} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 0)}_{\text{actually does U}} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 0)}_{\text{actually does D}} = 9 & \text{con l'azione R} \in \mathcal{A}(x) \\ \underbrace{0.8 \cdot (-1 + \gamma \cdot 0)}_{\text{goes in } S_{2,3}} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 0)}_{\text{actually does D}} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 0)}_{\text{actually does U}} = -1 & \text{con l'azione L} \in \mathcal{A}(x) \\ \underbrace{0.8 \cdot (-1 + \gamma \cdot 0)}_{\text{does in } S_{3,2}} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 25)}_{\text{actually goes L}} + \underbrace{0.1 \cdot (-1 + \gamma \cdot 0)}_{\text{actually goes R}} = 0.25 & \text{con l'azione D} \in \mathcal{A}(x) \end{cases}$$

$= 9$ con l'azione R

Abbiamo quindi che $v_1(x) = 9$ scegliendo l'azione R, quindi impostiamo $\pi_1(x) = R$.

Notiamo che tutti i reward sono costanti a -1 e quindi possiamo riscrivere la Bellman back-up function diminuendo i calcoli da svolgere:

$$\begin{aligned} v_{k+1}(s) &= \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s' | s, a) (R(s, a, s') + \gamma v_k(s')) && \text{Bellman back-up function} \\ &= \max_{a \in \mathcal{A}(s)} \left(\sum_{s'} p(s' | s, a) R_{const} + \sum_{s'} p(s' | s, a) \gamma v_k(s') \right) && \text{se } \forall s, a, s', R(s, a, s') = R_{const} \\ &= R_{const} + \max_{a \in \mathcal{A}(s)} \sum_{s'} p(s' | s, a) \gamma v_k(s') && \text{estrazione della costante} \end{aligned}$$

e con questa formula più efficiente nel nostro caso calcoliamo la seconda iterazione

$$v_1(x) = -1 + \max_{a \in \mathcal{A}(x)} \begin{cases} \underbrace{0.8 \cdot \gamma \cdot 9}_{v_1(x)} + \underbrace{0.1 \cdot \gamma \cdot 0}_{v_1(S_{2,3})} + \underbrace{0.1 \cdot \gamma \cdot 25}_{v_1(+25)} = 4.85 & \text{con l'azione U} \in \mathcal{A}(x) \\ \underbrace{0.8 \cdot \gamma \cdot 25}_{\text{goes in } +25} + \underbrace{0.1 \cdot \gamma \cdot 9}_{\text{actually does U}} + \underbrace{0.1 \cdot \gamma \cdot 0}_{\text{actually does D}} = 10.45 & \text{con l'azione R} \in \mathcal{A}(x) \\ \underbrace{0.8 \cdot \gamma \cdot 0}_{\text{goes in } (2,3)} + \underbrace{0.1 \cdot \gamma \cdot 0}_{\text{actually does D}} + \underbrace{0.1 \cdot \gamma \cdot 9}_{\text{actually does U}} = 0.45 & \text{con l'azione L} \in \mathcal{A}(x) \\ \underbrace{0.8 \cdot \gamma \cdot 0}_{\text{goes in } (3,2)} + \underbrace{0.1 \cdot \gamma \cdot 25}_{\text{actually does L}} + \underbrace{0.1 \cdot \gamma \cdot 0}_{\text{actually does R}} = 1.25 & \text{con l'azione D} \in \mathcal{A}(x) \end{cases}$$

$= 9.45$ con l'azione R

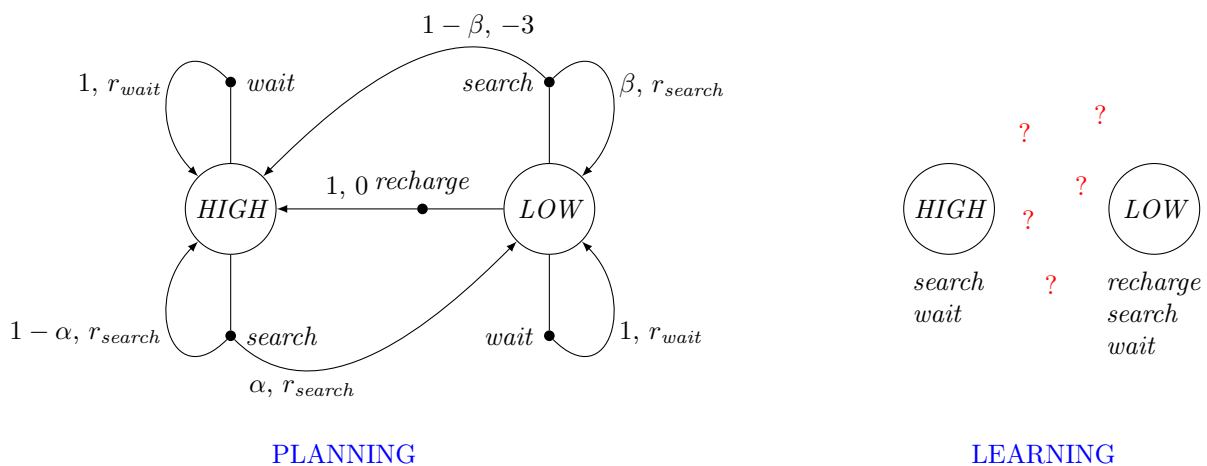
Abbiamo quindi che $v_2(x) = 9.45$ scegliendo l'azione R, quindi impostiamo $\pi_2(x) = R$. Ovviamente questo valore non è quello finale effettivo, in quanto non abbiamo considerato i valori degli stati vicini (non ne abbiamo calcolato il valore alla prima iterazione infatti) e soprattutto abbiamo troncato l'esecuzione iterata; in ogni caso notiamo l'efficienza di Value Iteration, che con pochi passi è già riuscita a raggiungere un punto pressoché fisso per lo stato da noi considerato.

7 | Learning

All'inizio del corso abbiamo visto le 4 categorie di agenti possibili e ci siamo soffermati sui *simple reflex agents*, per i quali abbiamo definito i comportamenti in ambienti noti (ricerca), con vincoli sui possibili risultati (reti a vincoli) e in ambienti incerti (Markov Decision Processes). Ora vogliamo distaccarci dalla parte attiva dell'insegnamento all'agente, lasciando che esso apprenda la scelta migliore da fare in maniera autonoma, sbagliando e correggendosi, costruendo effettivamente un *learning agent*.

7.1 Reinforcement Learning

Poniamoci nel caso in cui conosciamo gli stati ma non il modello di transizione $T(s, a, s')$ e nemmeno il reward $R(s, a, s')$: abbiamo bisogno di stimare le probabilità delle transizioni e scoprire i reward in modo più accurato possibile, non facendo *planning* come coi MDP ma facendo *learning*, ovvero *eseguendo delle azioni* e scoprendo il mondo e le azioni migliori da compiere man mano.



Per permettere all'agente di imparare possiamo utilizzare due approcci:

model-based: nel quale vogliamo che l'agente stimi il **MDP** dell'ambiente, per poi utilizzare le tecniche di risoluzione viste nel capitolo precedente; in questo modo, grazie al MDP

- evito di ripetere il compiere azioni pessime e l'andare in stati svantaggiosi;
- ho meno passi di esecuzione poiché conosco l'ambiente, quindi è più efficiente;
- utilizzo meglio i dati dell'ambiente, poiché interagisco poco con l'ambiente (dopo aver costruito il MDP le percezioni sono del tutto superflue).

model-free: nel quale vogliamo che l'agente stimi direttamente la **Q-function** e la **policy**; in questo modo

- non siamo dovuti a costruire un modello, quindi l'approccio è molto semplice;
- non ci sono i bias presenti quando invece costruiamo il modello, quindi agiamo esattamente di conseguenza all'ambiente e non in base alle previsioni del modello.

Esempio

Per stimare l'età di un gruppo di persone posso usare l'approccio

model-based: utilizzo come modello la distribuzione di probabilità delle età $\mathbb{E}[A] = \sum_a P(a) \cdot a$, quindi stimo $P(a)$ con $\hat{P}(a) = \frac{\#a}{N}$ dove $\#a$ è il numero di persone che hanno età a mentre N il numero di persone in totale. In questo modo abbiamo ottenuto $\mathbb{E}[A] \approx \sum_a \hat{P}(a) \cdot a$, che è una buona stima dell'ambiente poiché abbiamo usato il modello corretto;

model-free: non stimo assolutamente niente, infatti vado semplicemente a chiedere alle N persone

la loro età, quindi $\mathbb{E}[A] \approx \frac{1}{N} \sum_i a_i$ dove a_i è l'età della i -esima persona. In questo modo l'ambiente è descritto bene poiché abbiamo utilizzato i dati reali.

7.1.1 Approccio model-based

Dato che vogliamo stimare $\mathbb{E}[f(x)] = \sum_x P(x)f(x)$ per una certa funzione $f(x)$, possiamo raccogliere k sample $x_i \sim P(x)$, ovvero distribuiti secondo $P(x)$, e calcolare $\hat{P}(x) = \frac{\#x_i}{k}$. Nel nostro caso vogliamo stimare le probabilità del modello di transizione $T(s, a, s')$, quindi i nostri sample saranno sequenze stato-azione-successore $s_0, a_0, s_1, a_1, s_2, \dots$ di lunghezza imposta o che raggiungono uno stato terminale dette **learning episode** e la stima sarà

$$\hat{T}(s, a, s') = \frac{\#(s_t=s, a_t=a, s_{t+1}=s')}{\#(s_t=s, a_t=a)}$$

La reward function $R(s, a, s')$ invece viene stimata con

$$\hat{R}(s, a, s') = \frac{\sum R(s_t=s, a_t=a, s_{t+1}=s')}{\#(s_t=s, a_t=a, s_{t+1}=s')}$$

poiché potrebbe non essere deterministica, quindi facciamo la media dei reward degli episodi.

Esempio

Per il solito caso del robot ricicla lattine, abbiamo gli episodi

$$E1 = (L, R, H, 0), (H, S, H, 10), (H, S, L, 10)$$

$$E2 = (L, R, H, 0), (H, S, L, 10), (L, R, H, 0)$$

$$E3 = (H, S, L, 10), (L, R, H, 0), (H, S, L, 10)$$

e vogliamo stimare $\hat{T}(s, a, s')$ e $\hat{R}(s, a, s')$. Dobbiamo procedere caso per caso:

$$\hat{T}(L, R, H) = \frac{\#(L, R, H)}{\#(L, R)} = \frac{4}{4} = 1.0$$

$$\hat{T}(H, S, H) = \frac{\#(H, S, H)}{\#(H, S)} = \frac{1}{5} = 0.2$$

$$\hat{T}(L, R, L) = \frac{\#(L, R, L)}{\#(L, R)} = \frac{0}{4} = 0.0$$

$$\hat{T}(H, S, L) = \frac{\#(H, S, L)}{\#(H, S)} = \frac{4}{5} = 0.8$$

e così via per tutte le altre combinazioni. Per la reward function avremo invece

$$\hat{R}(L, R, H) = \frac{\sum R(L, R, H)}{\#(L, R, H)} = \frac{0}{4} = 0$$

$$\hat{R}(H, S, H) = \frac{\sum R(H, S, H)}{\#(H, S, H)} = \frac{10}{1} = 10$$

$$\hat{R}(L, R, L) = \frac{\sum R(L, R, L)}{\#(L, R, L)} = \frac{0}{0} \rightarrow ?$$

$$\hat{R}(H, S, L) = \frac{\sum R(H, S, L)}{\#(H, S, L)} = \frac{40}{4} = 10$$

Con quanto appena visto possiamo ridurre l'intero approccio al seguente algoritmo, nel quale agiamo direttamente in base alla policy, quindi non c'è nessuno stimolo alla **exploration**, ovvero scoprire stati dell'ambiente.

Algorithm 19 Approccio model-based al Reinforcement Learning

Require: A, S, S_0

Ensure: $\hat{T}, \hat{R}, \hat{\pi}$

- 1: inizializza arbitrariamente $\hat{T}, \hat{R}, \hat{\pi}$
 - 2: **repeat**
 - 3: esegui $\hat{\pi}$ per un *learning episode* // l'agente agisce nel mondo
 - 4: acquisisci una sequenza di tuple $\langle (s, a, s', r) \rangle$ // sono gli episodi
 - 5: aggiorna \hat{T} e \hat{R} in base alle tuple $\langle (s, a, s', r) \rangle$ // con le formule viste sopra
 - 6: data la dinamica corrente, calcola la policy // con Value Iteration oppure Policy Iteration
 - 7: **until** non incontriamo la condizione di terminazione
-

7.1.2 Approccio model-free

Dato che vogliamo stimare $\mathbb{E}[f(x)] = \sum_x P(x)f(x)$ per una certa funzione $f(x)$, possiamo lavorare direttamente sugli N episodi $x_i \sim P(x)$ che abbiamo raccolto finora, i quali seguiranno appunto la distribuzione $P(x)$ e quindi appariranno con la frequenza giusta, quella dettata dall'ambiente; avremo quindi $\mathbb{E}[f(x)] \approx \frac{1}{N} \sum_i f(x_i)$.

L'idea è di ottenere la value function $v(s)$ data la policy $\pi(s)$, che dobbiamo stimare, quindi per ogni stato s

1. esegui π per ottenere alcuni episodi;

2. ogni volta che s è presente nell'episodio, somma i reward scontati dei sample successivi;
3. media le somme sul numero dei sample per quello stato.

Esempio

Per il solito esempio del robot ricicla lattine, con $\gamma = 1$, abbiamo usato la policy per ottenere gli episodi

$$E1 = (L, R, H, 0), (H, S, H, 10), (H, S, L, 10)$$

$$E2 = (L, R, H, 0), (H, S, L, 10), (L, R, H, 0)$$

$$E3 = (H, S, L, 10), (L, R, H, 0), (H, S, L, 10)$$

e ora vogliamo calcolare $v(s)$ per ogni stato. Mostriamo solo il caso per $v(L)$: per ogni sample che contiene $s = L$ andiamo a sommare da quel punto fino alla fine dell'episodio tutto il reward accumulato

$$\text{in } E1 \text{ abbiamo } (L, R, H, 0) \text{ quindi } \gamma^0(0) + \gamma^1(10) + \gamma^2(10) = 20$$

$$\text{in } E2 \text{ abbiamo } (L, R, H, 0) \text{ quindi } \gamma^0(0) + \gamma^1(10) + \gamma^2(0) = 10$$

$$\text{in } E2 \text{ abbiamo anche } (L, R, H, 0) \text{ quindi } \gamma^0(0) = 0$$

$$\text{in } E3 \text{ abbiamo } (L, R, H, 0) \text{ quindi } \gamma^0(0) + \gamma^1(10) = 10$$

quindi in totale abbiamo 40. Ora facciamo la media sul numero di sample che hanno $s = L$ e otteniamo $v(L) = \frac{40}{4} = 10$.

7.1.2.1 Temporal Difference Learning

Con l'idea precedente però non abbiamo sfruttato il fatto che stiamo supponendo (ma non stimando) che il modello di fondo sia un MDP e quindi, **data la policy** π , la value function v deve soddisfare anche l'equazione di Bellman

$$v_{\pi}(s) = \sum_{s'} T(s, \pi(s), s') (R(s, \pi(s), s') + \gamma v_{\pi}(s'))$$

e possiamo quindi utilizzarla per migliorare la stima di $v(s)$. Aniché andare semplicemente a sommare le reward dei sample, andiamo a sommare anche la stima per lo stato successivo, effettivamente quindi applicando l'equazione di Bellman: in questo modo eseguiamo una media mobile, in cui ogni sample varrà

$$sample = R(s, \pi(s), s') + \gamma v_{\pi}(s')$$

e andremo ad aggiornare $v_{\pi}(s)$ di conseguenza ad *ogni sample, ricordando l'esperienza passata* con

$$v_{\pi}(s) \leftarrow (1 - \alpha)v_{\pi}(s) + \alpha(sample)$$

Così facendo descriviamo $v_{\pi}(s)$ con una **differenza temporale** data da $sample - v_{\pi}(s)$, infatti

$$v_{\pi}(s) \leftarrow v_{\pi}(s) + \alpha(sample - v_{\pi}(s))$$

dove α è il fattore di **learning rate**, che deve decrescere col passare del tempo affinché la media converga, quindi semplicemente viene impostato ad $\alpha_n = \frac{1}{n}$: questo perché inizialmente non abbiamo alcuna esperienza e quindi c'è il bisogno di imparare molto dai sample, ma col passare del tempo avremo sviluppato un'esperienza sufficiente per considerare via via sempre meno i nuovi sample, poiché già li conosco bene.

Nella sua forma completa quindi ad ogni sample useremo la formula

$$v_{\pi}(s) \leftarrow (1 - \alpha)v_{\pi}(s) + \alpha(R(s, \pi(s), s') + \gamma v_{\pi}(s'))$$

che è simile all'equazione di Bellman, ma nella quale non è presente il modello di transizione poiché a noi ignoto e sostituito con le nostre esperienze pregresse.

Esempio

Con $\gamma = 1$ e $\alpha = 0.5$ (per questo esempio non lo facciamo decrescere), **abbiamo usato una policy** per

ottenere i *sample* (non parliamo più di episodi poiché calcoliamo $v(s)$ ad ogni nuovo sample)

$$\textcircled{1} (L, R, H, 0), \textcircled{2} (H, S, H, 10), \textcircled{3} (H, S, L, 10), \textcircled{4} (L, R, H, 0), \dots$$

e ora vogliamo calcolare $v(s)$. Utilizziamo la formula di update vista sopra

$$v_{\pi}(s) \leftarrow (1 - \alpha)v_{\pi}(s) + \alpha(R(s, \pi(s), s') + \gamma v_{\pi}(s'))$$

e mostriamo l'evoluzione di $v(s)$ scrivendone la traccia dei valori ottenuti a lato e ovviamente considerando nei calcoli sempre l'ultimo arrivato (poiché sovrascrive i precedenti).

$$\textcircled{1} v(L) = (1 - 0.5) \cdot 0 + 0.5 \cdot (0 + 1 \cdot 0) = 0$$

$$\textcircled{2} v(H) = (1 - 0.5) \cdot 0 + 0.5 \cdot (10 + 1 \cdot 0) = 5$$

$$\textcircled{3} v(H) = (1 - 0.5) \cdot 5 + 0.5 \cdot (10 + 1 \cdot 0) = 7.5$$

$$\textcircled{4} v(L) = (1 - 0.5) \cdot 0 + 0.5 \cdot (0 + 1 \cdot 7.5) = 3.75$$

$$\boxed{v(L)} \rightarrow \emptyset \xrightarrow{\textcircled{1}} \emptyset \xrightarrow{\textcircled{4}} 3.75$$

$$\boxed{v(H)} \rightarrow \emptyset \xrightarrow{\textcircled{2}} \emptyset \xrightarrow{\textcircled{3}} 7.5$$

Man mano che altri sample arriveranno i valori andranno a raffinarsi sempre di più, convergendo.

7.1.2.2 Q-Learning

Con il temporal difference learning abbiamo utilizzato un'assunzione molto pesante, ovvero di *avere già* una policy, ma nel caso generale non la abbiamo, e vogliamo quindi costruirla usando la value function. Ci rendiamo conto che non è possibile usare solamente v poiché non abbiamo la certezza della azioni che compiamo, e quindi abbiamo bisogno di associare il valore ottenuto all'azione che mi ha permesso di ottenerlo, così da prendere la decisione migliore.

Ci viene utile quindi una approssimazione $Q(s, a)$ della **quality function**, che vogliamo essere nella sua stessa forma, quindi

$$Q(s, a) = \sum_{s'} T(s, a, s') (R(s, a, s') + \gamma v(s'))$$

poiché stimando $Q(s, a)$ possiamo poi estrarre la migliore azione utilizzando $\pi(s) = \arg \max_a Q(s, a)$, cosa impossibile da fare se utilizziamo $v(s)$. Questo da luogo al metodo detto **Q-Learning**, nel quale stimo $Q(s, a)$ e costruisco la policy su di essa: ricordando che

$$v(s) = \max_{a \in \mathcal{A}(s)} Q(s, a) = \max_{a \in \mathcal{A}(s)} \sum_{s'} T(s, a, s') (R(s, a, s') + \gamma v(s'))$$

andiamo a definire l'analogo dell'equazione di Bellman per le quality function, ottenendo

$$Q(s, a) = \sum_{s'} T(s, a, s') (R(s, a, s') + \gamma \max_{a' \in \mathcal{A}(s')} Q(s', a'))$$

ed esattamente come fatto per la value function possiamo eliminare T , poiché ignoto, impostando

$$sample = R(s, \pi(s), s') + \gamma \max_{a'} Q(s', a')$$

e ottenendo $Q(s, a)$ in base alle esperienze pregresse e ai nuovi sample, come fatto per il temporal difference learning, quindi

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha(R(s, a, s') + \gamma \max_{a'} Q(s', a'))$$

che differisce alla formula di update precedente solo per il $\max_{a'}$, il quale ci permette di ottenere la *migliore* quality function dello stato successivo, poiché dobbiamo considerare ogni azione possibile.

Il vero vantaggio del Q-Learning è che la policy ottenuta è *sempre* ottimale, a patto che

- l'esplorazione sia sufficiente (devo imparare da molti sample **distinti**);
- il learning rate α sia abbastanza piccolo...
- ma non decresca troppo velocemente.

e la cosa migliore è che il Q-Learning trova la policy ottima in **off-policy learning**, cioè senza seguirla alla lettera, imparando quali sono le azioni migliori da fare anche quando si decide di non eseguire proprio quelle.

Algorithm 20 Q-Learning

```

1: inizializza  $Q(s, a)$  arbitrariamente  $\forall s \in \mathcal{S}, a \in A(s)$  e forza  $Q(\text{STATO\_TERMINALE}, \cdot) = 0$ 
2: for  $n\text{-episodes}$  do
3:   inizializza lo stato iniziale  $S$  // riporta l'agente allo stato di partenza
4:   repeat
5:     scegli l'azione  $A$  da eseguire in  $S$  in base a  $Q$ 
6:     esegui l'azione  $A$ , ricevi il reward  $R$  e osserva il nuovo stato  $S'$ 
7:      $Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha(R + \gamma \max_a Q(S', a))$ 
8:      $S \leftarrow S'$ 
9:   until  $S$  è terminale // oppure anche raggiunto un numero massimo di azioni
10: for all  $s \in \mathcal{S}$  do
11:    $\pi(s) \leftarrow \arg \max_{a \in A(s)} Q(s, a)$ 
return  $\pi$ 

```

Vediamo il termine $\max_a Q(S', a)$, che permettere sempre di scegliere il caso successivo migliore *senza considerare la policy attuale* e quindi l'algoritmo è appunto di *off-policy learning*.

7.1.2.3 SARSA

Questa è una alternativa *on-policy learning* di Q-Learning, cioè vengono compiute le azioni *solamente* in base alla policy che stiamo stimando, e quindi non si esplora mai più del necessario (notiamo infatti l'assenza del \max_a , sostituito dalla scelta dell'azione A' da parte della policy).

Algorithm 21 SARSA

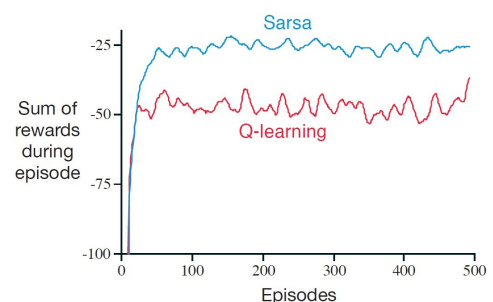
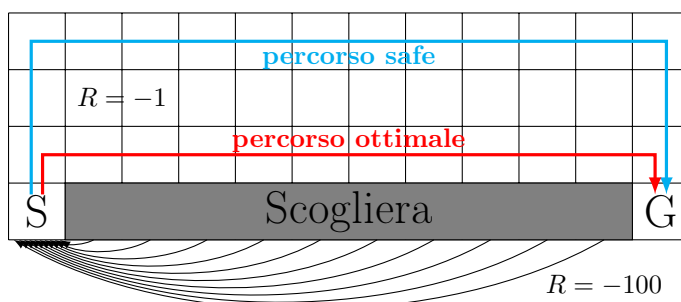
```

1: inizializza  $Q(s, a)$  arbitrariamente  $\forall s \in \mathcal{S}, a \in A(s)$  e forza  $Q(\text{STATO\_TERMINALE}, \cdot) = 0$ 
2: for  $n\text{-episodes}$  do
3:   inizializza lo stato iniziale  $S$  // riporta l'agente allo stato di partenza
4:   scegli l'azione  $A$  da eseguire in  $S$  in base a  $Q$ 
5:   repeat
6:     esegui l'azione  $A$ , ricevi il reward  $R$  e osserva il nuovo stato  $S'$ 
7:     scegli l'azione  $A'$  da eseguire in  $S'$  in base a  $Q$ 
8:      $Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha(R + \gamma Q(S', A'))$ 
9:      $S \leftarrow S', A \leftarrow A'$ 
10:  until  $S$  è terminale // oppure anche raggiunto un numero massimo di azioni
11: for all  $s \in \mathcal{S}$  do  $\pi(s) \leftarrow \arg \max_{a \in A(s)} Q(s, a)$ 
return  $\pi$ 

```

SARSA prende appunto il suo nome da fatto che utilizziamo una tupla (S, A, R, S', A') e ha la caratteristica di convergere verso l'ottimo $Q^*(s, a)$ se la policy converge alla policy greedy.

Studiamo entrambi i metodi poiché utilizzano filosofie molto diverse: il Q-Learning essendo off-line farà sempre il percorso migliore, scegliendo il percorso più breve, ma rischierà molto di più poiché essendo in un ambiente *incerto* ciò che fa non è deterministico (nell'esempio sotto, andando verso destra può invece cadere nella scogliera, ricominciando daccapo); SARSA invece fa sempre la scelta dettata dalla policy, che avendo imparato dagli episodi precedenti sa di dover stare il più distante possibile dagli stati svantaggiosi, e quindi intraprenderà percorsi più sicuri (nell'esempio sotto, sta il più possibile distante dalla scogliera). Con questo ragionamento, notiamo empiricamente che SARSA ha un valore di *reward per episode* più alto rispetto a Q-Learning, proprio perché sbaglierà meno spesso e quindi il reward medio sarà maggiore.



7.1.3 Dilemma Exploration vs Exploitation

Viste le considerazioni fatte prima, mi conviene seguire la strada che ho finora calcolato oppure rischio un pochino di più con la possibilità di trovare reward migliori? Per garantire la convergenza alla policy ottima dobbiamo esplorare un numero sufficiente di coppie stato-azione, quindi dobbiamo forzare l'agente a commettere degli errori ed eseguire azioni in modo incerto. Andiamo a vedere che cosa significa quel *in base a Q* che abbiamo lasciato in sospeso nei due algoritmi precedenti: data una certa Q per lo stato successivo, utilizziamo delle funzioni che impongono a volte di non fare proprio l'azione che avevamo in mente, così da permettere di ampliare un po' le nostre esperienze esplorando, senza seguire sempre il percorso dettato dalla logica (come se volessimo equipaggiare di "curiosità" l'agente):

ε -greedy: scegliamo l'azione a della policy in modo greedy (quindi la migliore possibile) nel $(1 - \varepsilon)$ dei casi, mentre nei restanti ε scegliamo equamente a caso tra tutte le altre azioni;

soft-max (o Boltzmann): , scegliamo l'azione a della policy con probabilità $p(a) = \frac{e^{Q(s,a)/T}}{\sum_{a'} e^{Q(s,a')/T}}$, dove T è un parametro *temperatura* (come nel Simulated Annealing); per T grande tutte le azioni sono equiprobabili, quindi esploriamo di più, mentre al diminuire di T siamo più attirati verso azioni con Q maggiore, quindi sfruttiamo di più quello che abbiamo già imparato (ovviamente rimane sempre una certa probabilità di non eseguire quello che vogliamo, ma diminuisce con la temperatura appunto).

Possiamo usare inoltre delle *funzioni di esplorazione*, che danno un bonus ogni volta che viene esplorato un nuovo stato: in questo modo siamo *ottimisti in faccia all'incertezza*, con l'idea di incentivare l'esplorazione di nuovi stati poiché sicuramente vantaggiosi (c'è un bonus appunto). In pratica l'idea è di aggiungere alla quality function dello stato successore un bonus che decresce in base al numero di volte che abbiamo visitato quello stato: usiamo la funzione $f(q, n) = q + k/n$, con k parametro fissato, per dare tale bonus e riscriviamo la funzione di update come

$$Q(S, A) \leftarrow (1 - \alpha)Q(S, A) + \alpha \left(R + \gamma \max_a f(Q(S', a), N(s', a')) \right)$$

dove $N(s', a')$ è semplicemente un contatore che conta le volte con cui abbiamo usato a' stando in s' .

7.1.4 Conclusioni

Abbiamo visto come possiamo creare un agente in modo che impari che cosa fare nell'ambiente in cui si trova proprio *mentre agisce*, quindi sbagliando e imparando dagli errori. Per farlo abbiamo utilizzato metodi *model-based*, che cercano di imparare la struttura dell'ambiente e poi valutare la policy, oppure metodi *model-free*, che stimano direttamente la qualità di fare una certa scelta in un certo stato, basandosi sulla propria "esperienza sul campo". Un altro aspetto importante è cercare di bilanciare esplorazione di nuovi stati, con possibilità di reward svantaggiosi, e sfruttamento di ciò che già ho imparato, che potrebbe tenermi però distante da stati più vantaggiosi che non ho esplorato; per ovviare a questo dilemma, adottiamo metodi (ε -greedy e soft-max) che "installano" indeterminatezza nella nostra policy: la maggior parte delle volte ci permettono di agire come da noi richiesto, facendoci sfruttare le esperienze, ma in altre istanze ci sviano, facendoci esplorare nuovi stati.

7.2 Deep Reinforcement Learning

Come abbiamo visto, per i MDP facciamo *planning* costruendo la policy in base alla value function (nel caso di Value Iteration), migliorata tramite la Bellman back-up function, mentre per il Reinforcement Learning usiamo la funzione di update per $Q(s, a)$: il problema principale del learning per come lo abbiamo visto è che c'è bisogno di una *tabella* per memorizzare $Q(s, a)$ poiché la quality value è valutata al variare di s ed a appunto; tale tabella diventa ingestibile per problemi con state-space continuo (ad esempio il problema del pendolo inverso) o comunque con un numero enorme di stati (ad esempio il gioco da tavolo Go oppure un videogioco della Atari).

L'idea è di utilizzare una policy *approssimata*, che riesce a gestire queste problematiche in uno di questi modi: ① approssimare direttamente la policy π , ② approssimare la value function v oppure ③ approssimare la quality function Q ; vedremo in dettaglio solo quest'ultimo approccio.

Per approssimare scomponiamo lo stato in un'insieme di features iniziali (nel caso del pendolo inverso avremo $s = (x, x', \theta, \theta')^T$ al tempo 0^- , per un gioco Atari i valori RGB di ogni pixel) e approssimiamo la funzione $Q(a, s)$ usando una linearizzazione $Q(a, s) \approx \sum_{i=1}^n w_{a,i} x_i = \mathbf{w}^T \mathbf{x}$ oppure una non-linearizzazione $Q(a, s) \approx g(\mathbf{x}; \mathbf{w})$, come nel caso delle *reti neurali*; fatto questo, il nostro compito è ora di stimare i parametri \mathbf{x} e \mathbf{w} .

7.2.1 Feed-Forward Artificial Neural Network

Costruiamo un DAG (Directed Acyclic Graph) che rappresenta una rete di **neuroni** che computano ognuno la funzione $h(\mathbf{w}^T \mathbf{x} + b)$, dove \mathbf{w} sono i pesi, \mathbf{x} sono gli input del neurone e b è un bias; la parte più importante è la **funzione di attivazione** h , che si occupa di filtrare non-linearmente i valori “utili” ottenuti linearmente dalla moltiplicazione matriciale. Le moltiplicazioni in sé non danno nessuna “intelligenza”, infatti se non ci fossero le funzioni h tutta la rete sarebbe collassabile ad un singolo nodo, quindi l’*intelligenza* della rete è data dal modo in essa cui filtra ciò che ritiene importante più che da come lo pesa. Vediamo alcune delle possibili funzioni di attivazione usate fino ad ora in letteratura (in particolare questo è un elenco di **squashing functions** poiché danno come risultati valori compresi tra -1 e 1 e quindi comprimono il valore calcolato dal nodo):

identità $h(x) = x$	soglia $h(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ -1 & \text{altrimenti} \end{cases}$	sigmoide $h(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	tangente iperbolica $h(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
gaussiana $h(x) = e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2}$	Rectified Linear Unit (ReLU) $h(x) = \max\{0, x\}$	soft computation (Softplus) $h(x) = \log(1 + e^x)$	

Una volta creata la rete la vogliamo poi **addestrare**, ovvero presentarle alcuni input \mathbf{x}_n di cui conosciamo l’expected output \mathbf{y}_n (questo insieme di coppie $(\mathbf{x}_n, \mathbf{y}_n)$ è detto **training set** per la rete) e, in base all’output $\bar{\mathbf{y}}_n = f(\mathbf{x}_n, \mathbf{w})$ della rete, misurare l’errore (detto **LOSS**) e minimizzarlo, aggiustando i pesi \mathbf{w} . Tra una vasta scelta di funzioni di errore, come esempio per ora possiamo misurare il LOSS con il *Least Square Error*

$$E(\mathbf{w}) = \sum_n E_n(\mathbf{w})^2 = \sum_n |f(\mathbf{x}_n, \mathbf{w}) - \mathbf{y}_n|^2$$

e per minimizzarlo aggiorniamo i pesi usando il **gradient descent** (come abbiamo visto in Hill-Climbing)

$$w_{ji} \leftarrow w_{ji} - \eta \frac{\partial E_n}{\partial w_{ji}}$$

Ovviamente dobbiamo aggiornare tutti i nodi della rete, poiché tutti sono responsabili di aver commesso l’errore, ma ognuno in quantità diversa, quindi abbiamo bisogno di associare al *gradient descent* anche l’algoritmo di **back-propagation**, il quale andrà a ridimensionare l’errore di ogni nodo in modo coerente a quello che è stato il suo contributo relativo all’interno della rete.

7.2.2 Deep Neural Networks

L’idea ora è di ampliare le reti neurali ad avere molti layer, sfruttando il fatto che utilizzando un numero infinito di gaussiane costruite ad-hoc possiamo ricostruire una qualsiasi funzione (è la controparte statistica della trasformata di Fourier): in questo modo, possiamo approssimare una qualsiasi funzione arbitrariamente bene a patto che la rete sia *sufficientemente grande*.

Teorema (Hornik et al., 1989, Cybenko, 1989)

Una feed-forward ANN con un layer di output lineare e almeno un hidden layer con una funzione di attivazione di “squashing” (sigmoide, tanh, gaussiana) può approssimare arbitrariamente bene *ogni* funzione a patto che la rete abbia *abbastanza* neuroni hidden. Con *ogni* si intende una funzione continua su un sottoinsieme limitato e chiuso di \mathbb{R}^n (in relazione con misurabilità di Borel).

Con questo capiamo che avere molti layer ci permette di approssimare funzioni complesse: da questo nascono le **Deep Neural Networks**, la cui unica peculiarità è avere almeno due hidden layer. Pur essendo molto simili alle ANN, le DNN hanno una potenza espressiva molto più ampia, in quanto hanno più interconnessioni con le quali “ragionare”, allo stesso modo con il quale gli esseri umani vengono considerati più intelligenti dei lombrichi.

7.2.3 Vanishing Gradient

Per applicare back-propagation dobbiamo eseguire la derivata della funzione di attivazione, ma le derivate delle funzioni descritte sopra ritornano sempre valori tra 0 e 1, il che significa che gli ultimi layer nella backpropagation avranno un’errore infimo a causa di queste graduali attenuazioni: per ovviare questo problema possiamo fare **pre-training**, **batch normalization**, **connections skip** o **utilizzare la ReLU**. La particolarità della ReLU infatti è di avere una derivata anch’essa non lineare: prima di 0 la derivata sarà costante 0, dopo invece sarà costante 1, il che significa che la funzione di errore sarà per i nodi ReLU un filtraggio binario a gradino, che non attenua mai la propagazione dell’errore, ma che lo propaga tale e quale oppure lo sopprime completamente.

7.2.4 Deep Q-Learning

Come abbiamo detto, in molti domini del mondo reale non possiamo rappresentare esplicitamente le funzioni chiave per il Reinforcement Learning, ovvero $\pi(s)$, $V(s)$, $Q(s, a)$, ma possiamo però approssimarle usando una rete neurale: descriviamo il procedimento che trasforma Q-Learning in **Deep Q-Learning**, un algoritmo di learning che approssima $Q(s, a)$ usando una DNN.

Andiamo innanzitutto ad approssimare $Q(s, a)$ con la funzione parametrica $Q_{\mathbf{w}}(s, a)$, che rappresenta essenzialmente la funzione $g(\mathbf{x}; \mathbf{w})$ descritta sopra, dove \mathbf{x} è la coppia (s, a) . A questo punto vogliamo minimizzare l'errore quadratico tra la **stima** e il **target**, ovvero minimizzare

$$E(\mathbf{w}) = \left(\overbrace{Q_{\mathbf{w}}(s, a)}^{\text{stima}} - \left(\overbrace{R(s, a, s') + \gamma \max_{a'} Q_{\bar{\mathbf{w}}}(s', a')}^{\text{target}} \right) \right)^2$$

dove $\bar{\mathbf{w}}$ è la matrice dei pesi reali, che però ovviamente non ho a priori e che via via vado a stimare nel tempo (vedremo come fare, è esattamente il solito problema che abbiamo affrontato anche con gli MDP e con il Q-Learning, ovvero che sto stimando qualcosa che non conosco basandomi su fatti di cui non so la correttezza, e vogliamo dimostrare che agire in questo modo converge comunque alla soluzione ottima). Lasciando da parte per un attimo questo problema, dobbiamo ora minimizzare $E(\mathbf{w})$, e quindi applichiamo il gradient descent usando la chain rule (molti termini sono costanti quindi la formula risulterà semplice)

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} = 2 \left(Q_{\mathbf{w}}(s, a) - \left(R(s, a, s') + \gamma \max_{a'} Q_{\bar{\mathbf{w}}}(s', a') \right) \right) \frac{\partial Q_{\mathbf{w}}(s, a)}{\partial \mathbf{w}}$$

dove il 2 è semplicemente dovuto al quadrato dell'errore ed essendo una costante moltiplicativa è ignorabile. Con questo abbiamo definito un approccio al Deep Q-Learning detto **Gradient Q-Learning**, che anziché utilizzare le tabelle sfrutta il gradiente, e ci permette di avvicinarci al mondo delle DNN.

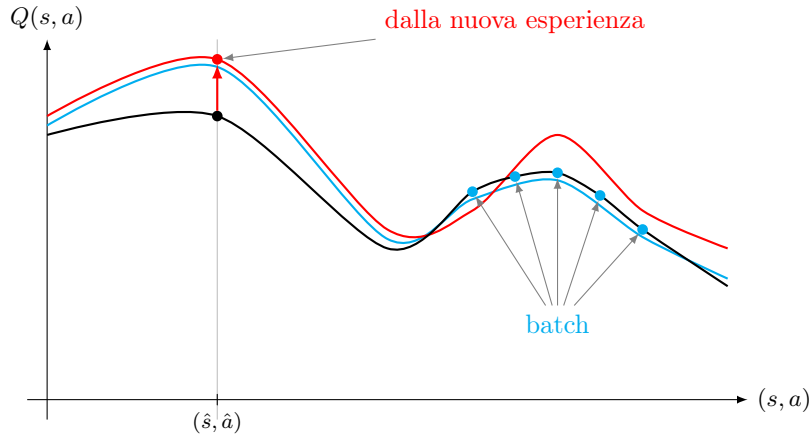
Algorithm 22 Gradient Q-Learning

- 1: inizializza i pesi \mathbf{w} casualmente con valori in $[-1, 1]$
 - 2: inizializza lo stato iniziale s
 - 3: **loop**
 - 4: seleziona ed esegui l'azione a , ricevi il reward r e osserva il nuovo stato s'
 - 5: calcola $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \leftarrow \left(Q_{\mathbf{w}}(s, a) - \left(r + \gamma \max_{a'} Q_{\bar{\mathbf{w}}}(s', a') \right) \right) \frac{\partial Q_{\mathbf{w}}(s, a)}{\partial \mathbf{w}}$
 - 6: aggiorna i pesi con $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$
 - 7: $S \leftarrow S'$
-

Notiamo che, come detto prima, abbiamo ignorato il 2, ma soprattutto che abbiamo utilizzato $Q_{\mathbf{w}}(s', a')$ anziché $Q_{\bar{\mathbf{w}}}(s', a')$, proprio perché non siamo a conoscenza di $\bar{\mathbf{w}}$. Questo ha intaccato la convergenza alla policy ottima? Non necessariamente! Ricordiamo che la convergenza è data da una esplorazione sufficiente dello spazio degli stati e dall'impostazione di un learning rate α che decresce non troppo velocemente. Abbiamo visto che adottando $\alpha_t = \frac{1}{t}$ abbiamo una decrescita buona, che applicata ad una linearizzazione $Q(s, a) \approx \mathbf{w}^T \mathbf{x}$ garantisce la convergenza, ma se applicata ad una non-linearizzazione $Q(s, a) \approx g(\mathbf{x}; \mathbf{w})$ non garantisce nulla (questo perché ora la correzione dell'errore di una sola esperienza (s, a) influisce su tutti i pesi \mathbf{w} , e non solo su quella specifica coppia), quindi abbiamo ancora un problema, non riusciamo a garantire la convergenza per Gradient Q-Learning! Anche se la convergenza non è garantita, possiamo però adottare delle tecniche per mitigare la possibile divergenza:

experience replay: si crea una memoria delle esperienze (s, a, r, s') già processate e ad ogni step, dopo aver processato una nuova esperienza, processiamo nuovamente un **batch** (un sottoinsieme) di esperienze prese dall'*experience buffer*, così da

- rafforzare il comportamento che avevamo prima della nuova esperienza e mitigare l'errore che possibilmente è stato da lei introdotto;
- ridurre la correlazione tra sample successivi (per il principio della località spaziale, l'esperienza successiva sarà in un intorno di questa, quindi rafforzare con un batch mitiga non solo un singolo punto ma tutta una porzione di curva che correla valori tra loro);
- riduce il numero di interazioni con l'ambiente (e quindi aumenta l'efficienza).



usare due reti differenti: utilizzo una **Q-network** $Q_{\mathbf{w}}(s, a)$ fare training e un'altra **target network** $Q_{\bar{\mathbf{w}}}(s, a)$ su cui basarmi; quest'ultima rimane per la maggior parte del tempo a valori fissi e viene aggiornata di tanto in tanto. Ad ogni nuova esperienza andremo quindi ad aggiornare la Q-network con l'equazione vista prima, quella con $Q_{\bar{\mathbf{w}}}(s', a')$, ovvero faremo

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha_t \left(Q_{\mathbf{w}}(s, a) - (r + \gamma \max_{a'} Q_{\bar{\mathbf{w}}}(s', a')) \right) \frac{\partial Q_{\mathbf{w}}(s, a)}{\partial \mathbf{w}}$$

e invece di tanto in tanto aggiorneremo anche la target network con $\bar{\mathbf{w}} \leftarrow (1 - \tau)\bar{\mathbf{w}} + \tau\mathbf{w}$, dove $0 < \tau < 1$ è un iperparametro solitamente molto piccolo (es. 0.005).

Quando uniamo **Gradient Q-Learning**, **experience replay** e l'utilizzo di una **target network** otteniamo l'algoritmo di **Deep Q-Network** o DQN, dove ricordiamo che “selezionare l'azione a ” significa utilizzare ε -greedy oppure softmax sulla Q corrente

Algorithm 23 Deep Q-Network

- 1: inizializza i pesi \mathbf{w} e $\bar{\mathbf{w}}$ casualmente con valori in $[-1, 1]$
 - 2: inizializza lo stato iniziale s
 - 3: **loop**
 - 4: seleziona ed esegui l'azione a , ricevi il reward r e osserva il nuovo stato s'
 - 5: aggiungi (s, a, r, s') all'experience buffer
 - 6: estrai un batch MB di esperienze dall'experience buffer
 - 7: **for all** $(\hat{s}, \hat{a}, \hat{r}, \hat{s}') \in MB$ **do**
 - 8: calcola $\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}} \leftarrow \left(Q_{\mathbf{w}}(\hat{s}, \hat{a}) - (\hat{r} + \gamma \max_{a'} Q_{\bar{\mathbf{w}}}(\hat{s}', \hat{a}')) \right) \frac{\partial Q_{\mathbf{w}}(\hat{s}, \hat{a})}{\partial \mathbf{w}}$
 - 9: aggiorna i pesi con $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$
 - 10: $S \leftarrow S'$
 - 11: ogni c step, aggiorna il target con $\bar{\mathbf{w}} \leftarrow (1 - \tau)\bar{\mathbf{w}} + \tau\mathbf{w}$
-

7.2.5 Conclusioni

La quality function $Q(s, a)$ non può essere rappresentata esplicitamente la maggior parte dei problemi reali, e quindi bisogna approssimarla in un qualche modo, ad esempio non-linearizzandola in una Deep Neural Network, nella quale usando *gradient descent* e *back-propagation* possiamo stimare la policy ottimale per l'ambiente. L'approccio di Deep Reinforcement Learning utilizza quindi una DNN per rappresentare $Q(s, a)$, e in particolare l'algoritmo Deep Q-Network risulta essere molto potente, quando abbiamo *spazio delle azioni discreto*; per spazi continui dobbiamo fare delle considerazioni extra, che qui non trattiamo.