Università degli Studi di Roma "La Sapienza"

Facoltà di Ingegneria
Dipartimento di Informatica e Sistemistica

Tesi di Laurea in Ingegneria Informatica

"Un Generatore di Cruciverba"

Candidato: Davide De Rosa

Relatore: Ing. Luigi Laura

Anno Accademico 2005/2006

Indice

Prefazione

Tutto il lavoro svolto è stato ispirato dall'interessante tesi di Sik Cambon Jensen e da cwc, un compilatore di cruciverba opensource scritto da Lars Christensen. Gli obiettivi sono due: il primo è sfruttare le basi teoriche per realizzare un compilatore più efficiente; il secondo è offrire un esempio reale di come i giochi siano spunti interessanti per chi studia l'Informatica. In linea con gli scopi profondamente didattici di questo studio, il software allegato è rilasciato sotto licenza GPL. Questa scelta permette, a chi fosse interessato, di usarne il codice come io ho potuto fare con il software altrui. La speranza è che gli sforzi di mantenere una struttura formale ma comprensibile siano apprezzati, sia in questo testo che nei sorgenti.

Nel primo capitolo viene descritto il gioco del cruciverba, con qualche cenno alle sue origini. Vengono delineati poi i punti di contatto di questo gioco enigmistico con il mondo informatico. Nell'ultimo paragrafo trova spazio anche il gioco del crucintarsio.

Con il secondo capitolo si introducono i Problemi di Soddisfacimento di Vincoli, con relativi algoritmi risolutivi ed alcuni esempi.

Nel terzo capitolo si forniscono alcuni modelli per la creazione di cruciverba tramite quanto esposto formalmente nel secondo capitolo.

Il quarto capitolo è di supporto alla lettura dei sorgenti del compilatore di cruciverba realizzato. In esso vengono spiegate le classi principali e documentati i parametri del programma.

Per finire nel quinto capitolo vengono fatte le considerazioni conclusive su come il software può rapportarsi ad applicazioni reali in termini di efficienza e qualità dei risultati.

Ringraziamenti

Ringrazio senz'altro la mia famiglia, per aver creduto in me quando ero io il primo a non farlo.

Ringrazio il prof. Luigi Laura che mi ha seguito nel corso dello stage con la massima disponibilità e cordialità.

Ringrazio il mio carissimo amico Ponch che ha sempre saputo dare grande valore ad ogni mio piccolo progetto, anche (e soprattutto) quello più inutile.

Ringrazio Ismone per tutte le giornate matte che hanno reso questi ultimi 2 anni i più colorati della mia vita.

Ringrazio particolarmente Laura e Francesco che mi hanno sopportato quasi quotidianamente su MSN.

Ringrazio gli amici con cui ho condiviso questi anni dell'Università: quelli che vedo più spesso (FoNzO, JohnCa ed Andrea), quelli che vedo sempre meno (Fabrizio, Luigi, Federico, Nicola ed Emanuele) e quelli che purtroppo non vedrò più (Davide).

Ringrazio con affetto sincero tutti gli amici più stretti a cui per qualche mese ho rinunciato per scrivere questa tesi, ma che sicuramente non mancheranno alla cerimonia.

Ringrazio Mauro e Giuseppe del laboratorio "Paolo Ercoli" di Via Tiburtina perché con loro, i borsisti e gli studenti che ho conosciuto negli ultimi mesi mi sono davvero divertito.

Ringrazio infine Leonardo Lanni perché è stato una guida validissima all'inizio dello stage, anche se poi abbiamo intrapreso strade differenti.

r0x, 21 maggio 2007

Capitolo 1

Il cruciverba

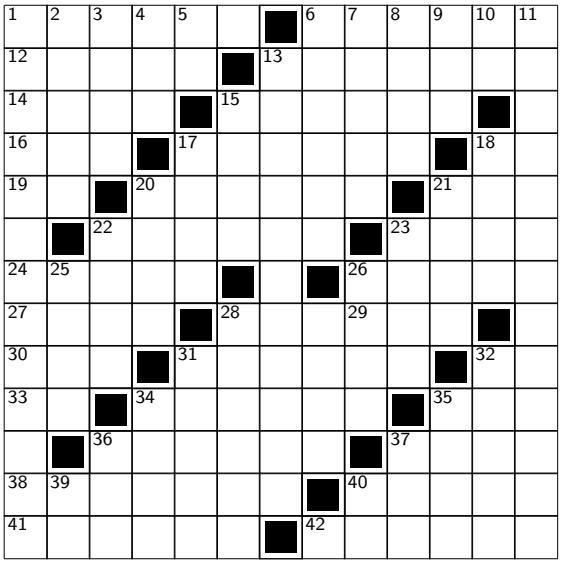
1.1 Scopo del gioco

Il **cruciverba** (o parole crociate) è uno dei più comuni giochi enigmistici. Si presenta come uno schema solitamente di forma quadrata o rettangolare, formato da caselle vuote —eventualmente numerate— o annerite. Lo scopo del gioco è quello di riempire le caselle vuote attraverso una lista di parole, dedotta a sua volta da una lista di definizioni che danno suggerimenti per la ricerca della parola associata.

Le parole possono seguire in genere due direzioni di riempimento, orizzontale o verticale, e rispettivamente da sinistra verso destra e dall'alto verso il basso. Il punto di inserimento di una parola è univocamente specificato dalla definizione; nella figura 1.1, ad esempio, la definizione "15 VERTICALE" va inserita a partire dalla casella con il numero 15 e dall'alto verso il basso. L'inserimento prosegue fino alla prima casella nera incontrata, quindi anche la lunghezza della parola è ben definita, ed è in questo caso pari a 4 lettere.

1.2 Storia del cruciverba

Domenica 21 dicembre 1913 sul supplemento del giornale *New York World* il giornalista di Liverpool Arthur Wynne pubblicò il primo *crossword puzzle* (cruciverba in inglese). Venne così inventato il cruciverba, in assoluto il più popolare e diffuso gioco di parole del mondo. Le prime parole crociate erano alquanto diverse dalla scacchiera quadrilatera, irregolare o simmetrica quanto alla posizione delle caselle nere, che siamo abituati a vedere. Il crossword di Wayne aveva piuttosto la forma di rombo (i cultori dicono "di diamante"), era molto semplice

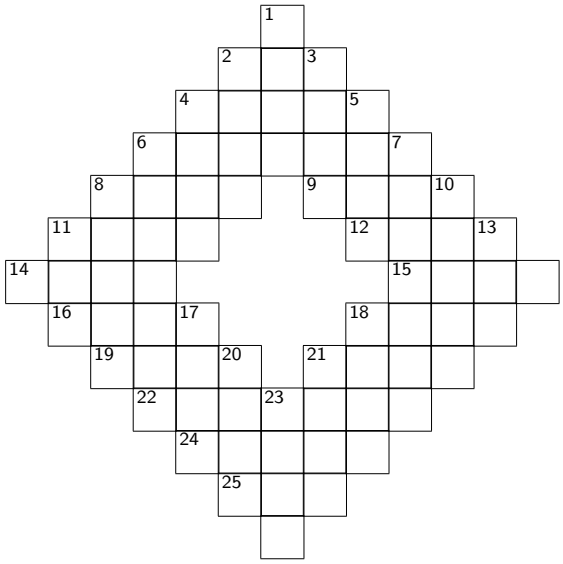


Orizzontali – 1 Una delle Eolie 6 Una Lauren di Hollywood 12 Varietà pregiata di calcedonio 13 Un Ermete tra i grandi del teatro 14 Autentici 15 Tenute agricole 16 Andate in breve 17 La provincia di Albenga 18 Il Conte cantautore (iniz.) 19 Vocali di peso 20 Sfocia nel Rio de la Plata 21 Precede il "Chi si vede!" 22 Consentono a pi'u di una persona di sedersi comodamente 23 L'insieme degli attori del film 24 Il regista di "C'eravamo tanto amati" 26 L'attore studia la propria 27 Sono fatali quelle di Bulgakov 28 Viaggia a cavallo di una scopa 30 Ha scritto "Lo scarabeo d'oro" 31 Una Rossana cantante 32 Sono doppie nel commissariato 33 Iniziali di Kipling 34 Una regione italiana 35 In una partita di tennis possono essere da tre a cinque 36 Insaccati 37 In un film, con amore e fantasia 38 Un essere... piccolo per antonomasia 40 L'Orietta della canzone 41 Nome della Fallaci 42 Il Gray di Wilde

Verticali – 1 Era la massima istituzione politica dell'URSS 2 È simile al finocchio 3 Avevano da quattro a sette corde 4 Un'imposta sugli immobili (sigla) 5 Coda di rondine 6 Giovanni Battista maestro di arte tipografica 7 Ne sono ghiotti i cavalli 8 Aspetto del volto 9 Califfo, genero di Maometto 10 Centro del Galles 11 Staterello tra Svizzera e Austria 13 L'ultimo minuto della partita di calcio 15 Gomma per suole 17 Fiume di Zagabria 18 Prefisso opposto ad ante 20 L'invenzione di Volta 21 Peso dell'imballaggio 22 Nel luogo in cui 23 Il Maggiore è Sirio 25 Grande navigatore inglese 26 Ruotano nel mulino a vento 28 Popolazione del Congo 29 Parti di un processo 31 Parte dell'intestino 32 Quella piperita è aromatica 34 La Venier 35 Veste femminile indiana 36 Sport di Stenmark 37 Segno tra i fattori 39 Simbolo dell'iridio 40 Estremità del berretto

Figura 1.1: cruciverba tratto dal giornale *Metro*

anche se si presentava come "esercizio per la mente" e soprattutto aveva i quadretti anneriti tutti raggruppati al centro. All'inizio il cruciverba era intitolato *word-cross puzzle*, ma per errore con la terza uscita i termini vennero invertiti e il nome anglosassone del gioco restò per sempre crossword puzzle. Il successo fu



Orizzontali – 4 What bargain hunters enjoy 6 A written acknowledgment 8 Such and nothing more 9 To cultivate 11 A bird 12 A bar of wood or iron 14 Opposed to less 15 What artists learn to do 16 What this puzzle is 18 Fastened 19 An animal of prey 21 Found on the seashore 22 The close of a day 24 To elude 25 The plural of is

Verticali – 1 To govern 2 Part of your head 3 A fist 4 A talon 5 Part of a ship 6 A day dream 7 Exchanging 8 What we all should be 10 To sink in mud 11 The fibre of the gomuti palm 13 A boy 17 A pigeon 18 One 20 A river in Russia 21 To agree with 23 An aromatic plant

Figura 1.2: cruciverba originale di Arthur Wynne (1913)

immediato, almeno negli Stati Uniti: lo stesso Wayne, che probabilmente non pensava di generare una tradizione, cominciò a pubblicare una delle sue griglie ogni settimana, almeno fino al 1916, data in cui fu chiamato alle armi nella prima guerra mondiale. Sussiste peraltro il fondato dubbio che il giornalista inglese abbia trovato ispirazione in una sorta di rompicapo di parole che era già in uso nel XIX secolo a Londra, il *Magic Square*, nel quale si dovevano arrangiare gruppi di parole in modo che si potessero leggere in verticale e in orizzontale. Le forme del puzzle erano diverse, anche circolari; infine Wynne si stabilizzò sul rettangolo. Il problema semmai erano gli errori di stampa, così esasperanti che ad un certo punto il creatore decise di lasciar perdere il suo gioco. Ma resistette una sola settimana alle proteste dei lettori.

La vera moda del cruciverba scoppiò comunque nell'aprile 1924, quando — per iniziativa di due editori esordienti— fu pubblicato negli USA il primo libro di schemi, completo di matita per le soluzioni. Risultato: alla fine dell'anno il volume —e due suoi emuli— erano ai primi tre posti nelle vendite di best-seller,

romanzi esclusi, con 400 mila copie smerciate; e quella serie continua tuttora. Tutti i quotidiani si diedero allora a pubblicare crossword puzzle, spesso offrendo ricompense ai solutori; solo l'altezzoso *New York Times* all'inizio rifiutò di accodarsi alla moda, cedendo solo nel 1942: oggi il suo cruciverba è considerato il migliore del genere (Bill Clinton è uno dei solutori più appassionati e abili). Ma non basta: in quegli anni Venti gli stilisti varano anche vestiti e gioielli ispirati alle caselle bianconere, le canzoni e il varietà citano il cruciverba, si organizzano campionati della specialità. È per questa febbre che alcuni preferiscono postdatare la nascita delle parole crociate al 1924. E su quest'onda il passatempo attraversa l'Oceano, cominciando ad essere pubblicato regolarmente sui quotidiani inglesi e francesi. Per dare un'idea della rapidità del boom anche in Europa: Demetrio Tolosani e Alberto Rastrelli - autori nel 1926 per Hoepli di una storia dell'Enimmistica in due volumi e, in quanto amanti dei giochi di parole classici, spregiatori del cruciverba troppo popolare —così si esprimevano riguardo al cruciverba:

> "il novissimo giuoco delle parole incrociate come un ciclone è passato attraverso il mondo, destando ovunque un interesse incomprensibile. Vero sì è che l'entusiasmo suscitato nei primi tempi è andato a poco a poco smorzandosi e crediamo di esser facili profeti se diciamo che fra breve tempo le parole incrociate o i crossword puzzle non saranno che un ricordo"

Nessuna profezia fu tanto errata: oggi infatti il cruciverba passa per essere il più diffuso gioco di parole del mondo, ed è presente ovunque eccetto i paesi dotati di lingua ideogrammatica. Nel 1928 comparvero, tra mille polemiche di natura teologica, anche i primi cruciverba in ebraico: la struttura delle parole crociate con quell'incrocio tra le lettere ricordava troppo la Croce. In Italia il passatempo apparve per la prima volta su *La Domenica del Corriere* esattamente l'8 febbraio 1925, quindi a immediato ridosso del boom del crossword puzzle americano, sotto il titolo di "Indovinello di parole incrociate"; era una smilza griglia di 5 caselle per 5, con sei definizioni. Già la settimana seguente, tuttavia, il cruciverba conquistò la copertina del diffusissimo settimanale, dove uno degli inconfondibili disegni di Achille Beltrame illustrava come a Londra la passione per le parole crociate fosse tale che in certe sale si giocava a risolverne uno schema gigante senza smettere di ballare. Lo stesso anno Mondadori stampava un libro di 50 cruciverba, con prefazione del linguista Fernando Palazzi e articoli dei letterati Emilio Cecchi e Valentino Bompiani. Nel 1931 —infine— nasceva la *Settimana Enigmistica*.

Così come per il telefono vi è una disputa se ad inventarlo sia stato Meucci o

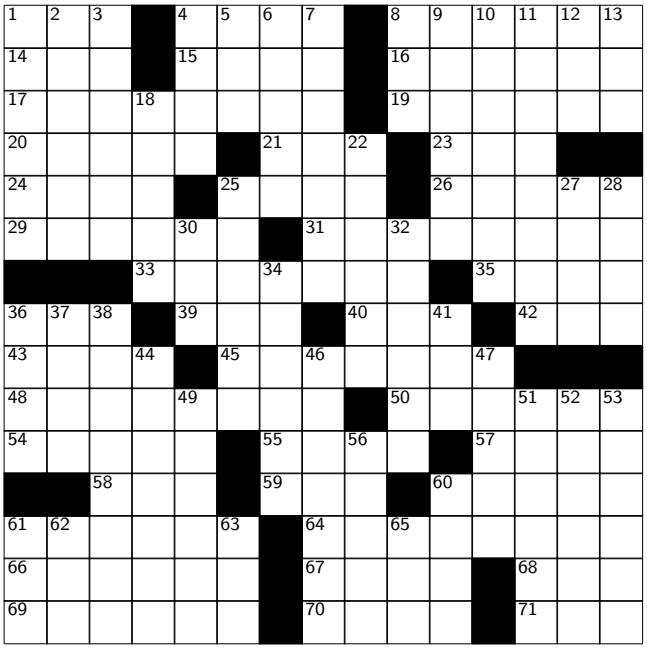


Figura 1.3: tipico cruciverba americano

Bell, anche per il cruciverba è in corso da anni una disputa italo-anglosassone. Pare infatti che un giornalista di Lecco, Giuseppe Airoldi in arte enigmistica "Inno Minato Monza", abbia pubblicato il suo cruciverba (4 caselle per 4) addirittura il 14 settembre 1890, sulle pagine del *Secolo Illustrato della Domenica*. Secondo un'altra versione della vicenda, il precursore della famosa scacchiera verbale sarebbe stato invece un americano, che nel 1875 sulla rivista *St. Nicholas* di New York stampò un "quadrato di parole" di 9 orizzontali per 9 verticali. Particolarissima la storia della diffusione del cruciverba in Sudafrica: l'inglese Victor Orille che nel 1915 —mentre era ubriaco al volante— ebbe un incidente d'auto in cui la moglie morì, chiese di essere inviato in prigione in Sudafrica, così da inasprire la sua pena; e laggiù per passatempo cominciò a disegnare quadrati di parole, che piacquero prima ai compagni di detenzione e poi al direttore del carcere, il quale li inviò a un giornale locale. Morale: Orille lasciò la prigione e divenne ricco, come nelle favole. E probabilmente passò la vecchiaia a risolvere cruciverba.

1.3 Il cruciverba in informatica

In rete è facile reperire software che trattano di cruciverba. Bisogna però effettuare una radicale distinzione fra software di risoluzione e di creazione di un

cruciverba. La difficoltà dei due problemi, per quanto correlati, ha origini piuttosto differenti. Infatti il solutore è vincolato a rispettare le definizioni fornite, mentre il creatore ha la totale libertà per la scelta dello schema, delle parole e delle definizioni. Illustriamo molto brevemente le differenze fra i software che si occupano di automatizzare i due processi.

1.3.1 Software di risoluzione

Questo caso rientra a pieno titolo nella tematica dell'Intelligenza Artificiale, e si può tranquillamente affermare che gli studi in questo ambito sono attivissimi. Anzi, più di qualcuno ha osservato che gli eredi di *Deep Blue* proseguiranno la sfida contro l'uomo proprio sul campo dei cruciverba.

La complessità di un software del genere è fortemente legata alla capacità del software stesso di "ragionare", e di giungere alle parole interpretando la semantica delle definizioni. Unitamente a questo obiettivo c'è la necessità di sperimentare efficienti algoritmi di ricerca, in grado di verificare la correttezza formale delle parole scelte all'interno dello schema (rispetto della lunghezza, rispetto degli incroci, presenza nel dizionario etc.). Quest'ultimo è un esempio di ciò che nel capitolo 2 verrà formalizzato come CSP.

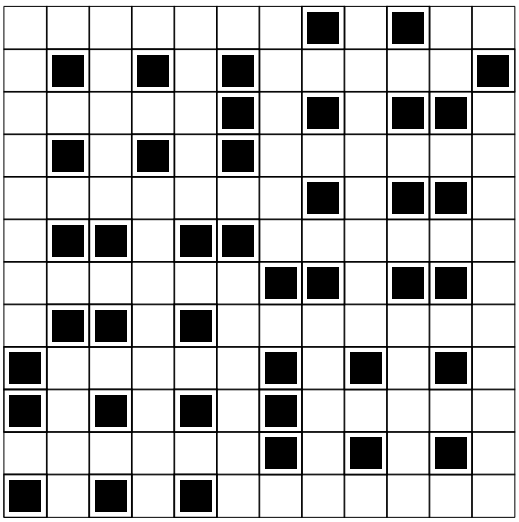
Un progetto a riguardo è *WebCrow* [11], sviluppato dalla facoltà di Ingegneria dell'Università di Siena. Uno studio approfondito delle idee alla base di *WebCrow* rivela l'elevata difficoltà del problema della risoluzione. È anche per questo motivo che il discorso della risoluzione non sarà oggetto di questo studio.

1.3.2 Software di creazione

La complessità, che nei software di risoluzione è principalmente incentrata sull'*automatic reasoning*, nei software di creazione si sposta decisamente sugli algoritmi di ricerca. La ricerca ha come scopo il riempimento di uno schema cruciverbale attraverso parole di un dizionario, garantendo che lunghezze ed incroci siano rispettati. Quindi anche in questo caso il problema è formulabile come un CSP. Non è tuttavia da escludere che anche un software di creazione possa operare delle scelte "intelligenti". Questa considerazione sarà fondamentale quando, formalizzando i CSP, si concluderà che il bruteforce (2.3.1) è computazionalmente improponibile.

1.4 Una variante: il crucintarsio

Se si esclude il problema delle definizioni, ma si decide invece di fornire direttamente le parole da inserire, il cruciverba assume l'aspetto del cosiddetto **crucintarsio** (in inglese *fill-in puzzle*). Naturalmente, scomparendo il vincolo delle definizioni, bisogna trasferire la difficoltà della risoluzione altrove: nel crucintarsio l'incognita diventa la posizione delle parole. È proprio per questo motivo che le caselle stavolta non sono numerate. Le parole sono raggruppate per lunghezza, ed il giocatore in genere tende a partire dalle parole più lunghe, perché più vincolanti (solitamente la parola più lunga è una sola) per gli inserimenti successivi. Un esempio di crucintarsio è mostrato nella figura 1.4.



2 lettere – ME MI
4 lettere – CEDE
5 lettere – CHILO COTTE ERIGI FLORA PIZZE TROVO TUONI ZAINI
6 lettere – ADULAI HACKER IRONIE OCCHIO OLANDA RISCHI
7 lettere – ERETICO OLFATTO RIANIMI RIPIENE
8 lettere – NOCCIOLO OTTURARE STECCATI
10 lettere – URBANESIMO

Figura 1.4: esempio di crucintarsio

Effettivamente, il software implementato in questo studio mirerà principalmente alla creazione di crucintarsi piuttosto che di cruciverba in senso stretto, poiché si tralascerà la generazione delle definizioni.

Capitolo 2

Cenni sui CSP

2.1 Introduzione

I **Constraint Satisfaction Problems** (= Problemi di Soddisfacimento di Vincoli) costituiscono una classe di problemi matematici fra le più ricche e studiate. Possiamo considerare un CSP come un problema in cui ad un insieme di **variabili** devono essere assegnati valori che rispettino dei **vincoli** imposti dal problema stesso. L'insieme dei valori ammissibili per una variabile è denominato **dominio** della variabile.

Anche una qualsiasi persona che non ne ha mai sentito parlare formalmente, si è sicuramente trovata di fronte ad un esempio, anche banale, di CSP. Se già ci si sofferma sui giochi, validi esempi sono il calcolo enigmatico, il cubo di Rubik, il gioco del 15, il Sudoku, le 8-Regine e così via. Nel Sudoku le variabili sono le caselle vuote a cui assegnare un numero da 1 a 9 (dominio), e i vincoli sono l'obbligo a scegliere tali numeri in modo che compaiano una volta sola nella stessa riga, nella stessa colonna e nello stesso sottoblocco 3x3 dello schema. Nelle 8-Regine le variabili potrebbero essere le posizioni delle regine nelle 8 righe distinte della scacchiera, con il vincolo che nessuna delle 8 regine attacchi (in orizzontale, verticale o diagonale) le altre 7.

Lo studio di metodi efficienti di risoluzione dei CSP è di particolare interesse per materie come la Ricerca Operativa, nella Teoria della Computabilità e nell'Intelligenza Artificiale. Sono però ben più vaste le applicazioni della teoria che ruota intorno ai CSP. Anche problemi che all'apparenza non sembrano correlati alla matematica o all'informatica, possono essere reinterpretati come CSP.

2.2 Formulazione

Definizione 2.1 (Variabile e Dominio) *Si definisce* variabile x *di un problema con relativo* dominio D *una incognita che può assumere un qualsiasi valore* $d \in D$.

È evidente che esistono CSP le cui variabili possono operare su domini reali, ma assumeremo in questo testo che essi siano insiemi discreti e finiti. Se consideriamo inoltre che una variabile può trovarsi in due stati, ovvero può avere o non avere un valore, possiamo introdurre un ulteriore dettaglio:

Definizione 2.2 (Assegnazione) *Sia* ε *il valore nullo. Una variabile* x *con relativo dominio* D *si dice assegnata se* $x = d$ *con* $d \in D$*; una variabile si dice non assegnata se* $x = \varepsilon$*. Dato un insieme di variabili* $X = \{x_1, x_2, \ldots, x_n\}$ *con relativi domini* $D = \{D_1, D_2, \ldots, D_n\}$*, sarà indicato con il termine* assegnazione *di* X *un qualsiasi stato* $V = \{v_1, v_2, \ldots, v_n\}$ *delle variabili di* X *definito come segue:*

$$\begin{cases} x_i = v_i & \text{per } i = 1, 2, \ldots, n \\ v_i \in D_i^0 = D_i \cup \{\varepsilon\} & \text{per } i = 1, 2, \ldots, n \end{cases}$$

Ci riferiremo con la stessa terminologia anche nel caso banale di una sola variabile. Per esempio "assegnazione $x = v$ con $v \in D \cup \{\varepsilon\}$". Nel corso del testo diremo inoltre che un'assegnazione è completa se:

$$v_i \neq \varepsilon \qquad \text{per } i = 1, 2, \ldots, n$$

e nulla se:

$$v_i = \varepsilon \qquad \text{per } i = 1, 2, \ldots, n$$

Indicheremo l'insieme di tutte le possibili assegnazioni di X con:

$$V^*(X) = D_1^0 \times D_2^0 \times \cdots \times D_n^0$$

che rappresenta il prodotto cartesiano di tutti i valori possibili (anche nulli) delle variabili x_i. L'insieme di tutte le possibili assegnazioni complete di X sarà indicato invece con:

$$V^+(X) = D_1 \times D_2 \times \cdots \times D_n$$

Definizione 2.3 (Vincolo) *Dati un insieme di variabili* $W = \{x_1, x_2, \ldots, x_r\}$ *e tutte le possibili assegnazioni* $V^*(W)$*, si definisce* vincolo c *una funzione booleana (o predicato) della forma:*

$$c : V^*(W) \longrightarrow \{0, 1\}$$

Data un'assegnazione $V' = \{v_1, v_2, \ldots, v_r\}$ *di* W*, si dice che il vincolo è* soddisfatto *da* V' *se* $c(V') = 1$*, in tal caso* V' *è* consistente*; il vincolo è* violato *da* V' *se* $c(V') = 0$*, in tal caso* V' *è* inconsistente*. Le variabili* $x \in W$ *si dicono coinvolte nel vincolo. L'insieme* W *è chiamato* ambito *di* c*, e la sua cardinalità definisce l'*arità *del vincolo. In particolare, un vincolo di arità 1 è detto unario, mentre di arità 2 è detto binario.*

Un vincolo si dice perciò binario se coinvolge 2 variabili, ma potrebbe non essere chiaro il significato di vincolo unario. Effettivamente, tale accezione potrebbe essere stata tralasciata. Questo perché un vincolo unario si può risolvere in un vincolo di dominio, e tale restrizione è intrinseca nella definizione di variabile. Consideriamo ad esempio il Sudoku, e partiamo dal requisito che le variabili siano numeri naturali, ovvero $D_i = \mathbb{N}$; il secondo passo è imporre il vincolo unario che tali numeri siano nell'intervallo 1–9, come previsto dal gioco. Il vincolo può essere eliminato agevolmente imponendo $D_i = \{1, 2, \ldots, 9\}$. La procedura di eliminazione dei vincoli unari porta i domini D_i al cosiddetto stato di **node consistency**.

Per concludere, forniamo un'altra interpretazione del concetto di vincolo:

Definizione 2.4 (Relazione associata) *Dato un vincolo binario* c *e i domini* $\{D_1, D_2, \ldots, D_r\}$ *delle variabili in esso coinvolte, si definisce* relazione associata *al vincolo* c *e si indica con* $\mathrm{rel}(c)$ *l'insieme:*

$$\mathrm{rel}(c) = \{\langle d_1, d_2, \ldots, d_r\rangle \in D_1 \times D_2 \times \cdots \times D_r : c(\{d_1, d_2, \ldots, d_r\}) = 1\}$$

delle assegnazioni ordinate $\langle d_1, d_2, \ldots, d_r\rangle$ *che soddisfano il vincolo.*

Per conformità con il concetto di relazione associata indicheremo talvolta un vincolo c fra r variabili $x_1, x_2, \ldots, x_r$ con la notazione $c = \langle x_1, x_2, \ldots, x_r\rangle$.

È importante osservare che la gran parte dei modelli di CSP si basa su vincoli esclusivamente binari, senza limitazione di generalità. Esistono infatti algoritmi di conversione di arità da vincoli n-ari a binari. Tali procedimenti si chiamano **binarizzazioni**, e permettono di ridurre l'intera classe dei CSP a quella dei CSP binari[1]. All'atto pratico, tuttavia, la conversione non sempre è conveniente. Resta comunque il fatto che in alcuni software per la risoluzione di CSP la restrizione a vincoli unicamente binari può essere un requisito fondamentale.

È possibile interpretare un CSP di fatto come un **grafo**[2], con tutti i vantaggi che ne scaturiscono. Tale grafo ha come nodi le variabili del problema, dove gli archi sono i vincoli che legano le variabili fra loro. Questa analogia permette di avvalersi degli algoritmi messi a disposizione dalla Teoria dei Grafi.

[1] non si trascuri a tal proposito l'assunzione di discretezza e limitatezza dei domini

[2] in generale, un grafo per vincoli binari ed un *ipergrafo* in caso di vincoli n-ari

Definizione 2.5 (CSP) *Un* CSP *è definibile come:*

1. *Un insieme di variabili* $X = \{x_1, x_2, \ldots, x_n\}$ *con rispettivi domini* $D = \{D_1, D_2, \ldots, D_n\}$
2. *Un insieme di vincoli* $C = \{c_1, c_2, \ldots, c_m\}$ *con rispettivi ambiti* $W = \{W_1, W_2, \ldots, W_m\}$ *tali che* $W_j \subseteq X$

L'insieme dei vincoli in cui una generica variabile x_i *è coinvolta sarà indicato con* $R_i \subseteq C$*. Condizione necessaria e sufficiente affinché un'assegnazione completa* $S = \{s_1, s_2, \ldots, s_n\}$ *di* X *sia soluzione del problema è che:*

$$c_j(S) = 1 \qquad \text{per } j = 1, 2, \ldots, m$$

cioè soddisfi tutti i vincoli.

Generalmente, la specifica di "qualità" di una soluzione non è intrinseca nel CSP, quindi affinché un'assegnazione sia soluzione del problema, è sufficiente che essa soddisfi tutti i vincoli. Il grado di bontà di una soluzione viene piuttosto elevato aumentando o perfezionando i vincoli del problema. Questo tipo di modello permette di mantenere un alto livello di generalizzazione, e di trattare così con gli stessi algoritmi problemi notevolmente eterogenei.

Nonostante la profonda matrice comune, vedremo però che per la risoluzione efficiente di un CSP non di rado è necessario sviluppare tecniche *ad hoc* strettamente correlate alla natura del problema. L'approccio euristico (2.4), seppur non esaustivo, è molto spesso quello più efficace.

2.3 Alcuni algoritmi risolutivi

Di seguito verranno descritti gli algoritmi più semplici e generali per la ricerca di una soluzione ammissibile per un CSP. Come accade quando ci si trova a parlare di algoritmi, anche in questo testo verrà adottata la notazione "O grande" di una funzione $f(n)$, o meglio $\mathcal{O}(f(n))$. Si darà per scontato nel corso di questo capitolo che il significato di "O grande" sia noto. La premessa appena fatta è necessaria per poter formulare in termini di $\mathcal{O}(f(n))$ la **complessità di caso peggiore** di un algoritmo. Nello pseudocodice invece si assumerà per semplicità che il CSP sia binario.

2.3.1 Bruteforce

L'approccio *naïve* ai CSP porta a pensare, in prima ipotesi, che per la ricerca di una soluzione ammissibile per il problema sia sufficiente esaminare tutte le

assegnazioni complete possibili delle variabili. Questo metodo di risoluzione è il cosiddetto **bruteforce** ed è il più primitivo per risolvere un CSP.

Algoritmo 1 BRUTEFORCE

Precondizioni: variabili X ($|X| = n$), vincoli C

Postcondizioni: restituisce una soluzione del problema, $NULL$ se problema inammissibile

```
1:  function BRUTEFORCE(X, C)
2:      for all S ∈ V⁺(X) do              /* assegnazioni complete possibili */
3:          SATISFIED ← true
4:          for all c ∈ C do
5:              if c(S) = 0 then
6:                  SATISFIED ← false     /* almeno un vincolo violato */
7:              end if
8:          end for
9:          if SATISFIED then
10:             return S                              /* soluzione */
11:         end if
12:     end for
13:     return NULL                       /* problema inammissibile */
14: end function
```

La convinzione fuorviante alla base del bruteforce, è che un computer sia in grado di effettuare calcoli in modo molto veloce, potenzialmente istantaneo. È proprio questo che metteremo in discussione. Consideriamo un insieme $X = \{x_1, x_2, \ldots, x_n\}$ di variabili di un CSP qualsiasi, siano $D = \{D_1, D_2, \ldots, D_n\}$ i rispettivi domini associati e sia $V^+(X)$ l'insieme di tutte le assegnazioni complete possibili. Assumiamo, per semplicità, che $|D_i| = d$ per $i = 1, 2, \ldots, n$. La cardinalità $|V^+(X)|$ è evidentemente pari a $|D_1| * |D_2| * \cdots * |D_n| = d^n$. Ciò significa che esaminare tutte le assegnazioni possibili ha costo computazionale esponenziale $\mathcal{O}(d^n)$, dove d è la dimensione dei domini[3]. Supponiamo che $n = 10$ e $d = 100$, cioè il CSP abbia 10 variabili ognuna con 100 valori possibili. Esistono CSP con centinaia, migliaia di variabili, eppure già in questo semplice scenario il numero di assegnazioni possibili è addirittura $d^n = 100^{10} = 10000000000000000000$. Ipotizziamo che la verifica di consistenza di un'assegnazione abbia costo lineare nel numero m dei vincoli, cioè abbia

[3]nel caso generale di domini con cardinalità differenti si considera d la cardinalità del dominio più grande

complessità $\mathcal{O}(m)$, e possa essere effettuata da un computer di prova in 1ns (che è un tempo di esecuzione molto basso): ad esaminare tutte le 100^{10} assegnazioni si impiegherebbero circa 3171 anni!

Il bruteforce non è certamente la tecnica più indicata per la risoluzione di un CSP. C'è da osservare però che neanche algoritmi attuali più avanzati riescono a superare l'ostacolo della complessità esponenziale, tant'è vero che la ricerca di metodi efficienti[4] di risoluzione dei CSP è una tematica ancora aperta nella Teoria della Computabilità[5].

2.3.2 Backtracking (BT)

Un sostanziale miglioramento del bruteforce consiste nel **backtracking**, che tiene in considerazione la natura vincolata del CSP. Il problema del bruteforce è che assegna un valore a tutte le variabili contemporaneamente, con successiva verifica in blocco di tutti i vincoli. Il BT invece parte da un'assegnazione nulla che ad ogni passo viene estesa cercando di assegnare (instanziare) una nuova variabile, ad esempio partendo da x_1 fino ad arrivare ad x_n. La scelta del valore della variabile viene operata in modo che la nuova assegnazione non sia inconsistente: se non esiste un valore che soddisfi questo requisito, l'instanziazione della nuova variabile fallisce e si ritorna a modificare il valore della variabile instanziata più di recente. Quello appena descritto è il cosiddetto **backtrack**. L'algoritmo termina quando l'assegnazione è completa o la prima variabile genera un backtrack.

Osserviamo dal punto di vista formale come il BT sia migliore del bruteforce. Siano $X = \{x_1, x_2, \ldots, x_n\}$ le variabili del CSP e $V = \{v_1, v_2, \ldots, v_n\}$ l'assegnazione corrente (consistente) in un qualsiasi passo dell'algoritmo; si supponga che le prime k variabili di X siano assegnate, cioè:

$$\begin{cases} v_i \neq \varepsilon & \text{per } i = 1, 2, \ldots, k \\ v_i = \varepsilon & \text{per } i = k+1, k+2, \ldots, n \end{cases}$$

Il BT cerca ora di assegnare la variabile x_{k+1}, e lo fa scegliendo un valore $v'_{k+1} \in D_{k+1}$ in modo che la nuova assegnazione $V' = \{v_1, v_2, \ldots, v'_{k+1}, \ldots, v_n\}$ sia ancora consistente. Se non esiste un valore di v'_{k+1} così definito, è inutile tentare di assegnare le restanti variabili oltre x_{k+1}, perché qualsiasi estensione di V' rimarrà in ogni caso inconsistente. Quindi l'algoritmo incontra un backtrack e ritira il valore di x_{k+1}. Siccome reinstanziare direttamente x_{k+1} non porta —come osservato— a nessuna soluzione, viene assegnato un nuovo valore

[4] un algoritmo è *efficiente* se ha complessità $\mathcal{O}(n^k)$ con $k = \text{cost}$ (polinomiale), o inferiore

[5] nella letteratura informatica i CSP sono un classico esempio di problemi *NP*-completi

Algoritmo 2 BACKTRACKING

Precondizioni: variabili X ($|X| = n$) con domini D e vincoli associati R, assegnazione nulla S, $i = 1$

Postcondizioni: se $true$, S soluzione del problema; se $false$, S nulla e problema inammissibile

```
1: function BACKTRACK(X, D, R, S, i)
2:     if i = n + 1 then
3:         return true                    /* assegnazione completa e soluzione */
4:     else
5:         for all d ∈ D_i do
6:             s_i ← d                                  /* instanziazione di x_i */
7:             if CHECK(R_i, S) then                          /* consistenza */
8:                 if BACKTRACK(X, D, R, S, i + 1) then  /* passo ricorsivo */
9:                     return true
10:                else
11:                    s_i ← ε                    /* instanziazione annullata */
12:                end if
13:            else
14:                s_i ← ε                        /* instanziazione annullata */
15:            end if
16:        end for
17:        return false                                        /* backtrack */
18:    end if
19: end function

20: function CHECK(R_i, S)
21:    for all c ∈ R_i do
22:        if c(S) = 0 then
23:            return false                /* almeno un vincolo violato */
24:        end if
25:    end for
26:    return true
27: end function
```

alla variabile più recente, in questo caso x_k. Così facendo non si preclude la consistenza delle estensioni successive di V. Il bruteforce è inefficiente perché cerca di completare anche assegnazioni parziali inconsistenti, ignorando di fatto

il backtrack.

Supponiamo di avere un CSP di 3 variabili x_1, x_2, x_3 con rispettivi domini $D_1 = \{2, 3, 5\}$, $D_2 = \{2, 7, 1\}$ e $D_3 = \{3, 1\}$; siano poi $c_1 =$ "$x_1 \neq x_2$", $c_2 =$ "$x_1 = 2x_3$" e $c_3 =$ "$x_2 = x_3$" i vincoli del problema. Il BT risolve il CSP in questo modo:

1. assegna x_1 cominciando dal primo valore di D_1, quindi $x_1 = 2$
2. assegna x_2, che viola c_1 per $x_2 = 2$ ma è consistente per $x_2 = 7$
3. tenta di assegnare x_3, ma $x_3 = 3$ viola c_2, c_3 e $x_3 = 1$ viola c_2: si ha un backtrack
4. riassegna x_2, stavolta con $x_2 = 1$
5. assegna $x_3 = 1$, perché $x_3 = 3$ violerebbe c_2: l'assegnazione $S = \{2, 1, 1\}$ è una soluzione del CSP

Nonostante sia evidente come all'atto pratico il BT sia molto migliore del bruteforce, asintoticamente la sua complessità rimane esponenziale. Questo accade perché nel caso peggiore il BT coincide proprio con il bruteforce. Siano x_1, x_2 le variabili di un CSP con domini $D_1 = \{1, 2, 3\}$ e $D_2 = \{3, 5\}$ e sia "$x_2 = 2x_1$" il vincolo che le lega; è banale constatare che il BT in questo caso esamina lo stesso numero di assegnazioni del bruteforce, cioè 6. Questo accade quando le prime $n - 1$ instanziazioni hanno sempre successo, mentre l'n-esima rileva sempre un backtrack.

Essendo basato sul concetto di stato, il BT è un algoritmo prettamente **ricorsivo**. Nel nostro caso, abbiamo identificato lo stato dell'algoritmo con l'assegnazione corrente. La condizione di base del BT di un CSP è evidentemente la completezza dell'assegnazione. Se il BT giunge ad un'assegnazione completa S, essa è automaticamente soluzione del CSP, perché la verifica di consistenza viene effettuata ad ogni passo dell'algoritmo. Se invece si incontra un backtrack nell'assegnazione della prima variabile del CSP, allora è chiaro che il CSP non ammette soluzioni.

I paragrafi successivi descrivono non tanto algoritmi a se stanti quanto raffinamenti del BT. Il BT semplice trova applicazione più che altro in ambito didattico, sebbene resti il punto di partenza di algoritmi più complessi.

2.3.3 Backjumping (BJ)

Il BT può essere perfezionato nella fase di backtrack, in particolare nella scelta della variabile da riassegnare. Nel caso illustrato nel paragrafo 2.3.2, se il backtrack avviene nel tentativo di instanziazione della variabile x_{k+1}, viene scelto un

nuovo valore per la variabile x_k. Non è detto però che, modificando x_k, esistano valori per x_{k+1} che non scaturiscano un backtrack sistematico. Sembra quindi ragionevole operare questa scelta con più attenzione. In particolare, conviene modificare la variabile instanziata più di recente fra quelle che influenzeranno le successive scelte di x_{k+1} (le variabili **condizionanti** di x_{k+1}). Un algoritmo di BT che sfrutta le considerazioni fatte finora è un esempio di **backjumping**.

Con il generico termine backjumping ci si riferisce sostanzialmente a tutti gli algoritmi di BT che in caso di backtrack —e attraverso assunzioni sullo stato della ricerca— possono "saltare" più di una variabile. Il requisito fondamentale è che il salto all'indietro non sia troppo esteso da scartare soluzioni possibili: è importante cioè che il BJ effettui sempre e solo un *safe jump*.

2.3.4 Forward checking (FC)

Il **forward checking** è il primo esempio di algoritmo **look-ahead**. Vengono denominati in questo modo gli algoritmi di BT che tengono conto dell'assegnazione corrente per poter influenzare gli stati successivi della ricerca. Negli algoritmi di questo tipo, ci si preoccupa di ridurre i domini ammissibili delle variabili non assegnate osservando come esse siano correlate a quelle già assegnate. Tale analisi è anche denominata **constraint propagation** in quanto "propaga" l'effetto dei vincoli del problema sulla base dello stato attuale del BT.

Concettualmente il FC è molto semplice. Dopo aver assegnato una variabile, l'algoritmo non fa altro che ridurre opportunamente i domini delle variabili che sono coinvolte nei suoi stessi vincoli. La riduzione si effettua eliminando tutti i valori che, affiancati alla nuova assegnazione, non soddisfano il vincolo. Se questa operazione porta ad almeno un dominio vuoto, allora è chiaro che la nuova assegnazione è inconsistente.

Supponiamo di avere un CSP di 3 variabili x_1, x_2, x_3 con domini $D_1 = \{2,3,5\}$, $D_2 = \{2,7,5\}$ e $D_3 = \{1,3\}$; siano poi $c_1 = \langle x_1, x_2\rangle =$ "$x_1 \neq x_2$" e $c_2 = \langle x_1, x_3\rangle =$ "$x_1 = 2x_3$" i vincoli del problema. Il BT tenta di assegnare x_1 cominciando dal primo valore di D_1, quindi $x_1 = 2$; il vincolo c_1 porta a scartare l'elemento $\{2\}$ da D_2, mentre c_2 porta a scartare l'elemento $\{3\}$ da D_3; i nuovi domini $D_2^{'} = \{7,5\}$ e $D_3^{'} = \{1\}$ sono non vuoti, quindi il BT può continuare assegnando x_2 e così via.

A fronte dell'overhead introdotto con il ricalcolo dei domini, il guadagno in termini di velocità di esecuzione è particolarmente rilevante. Il miglioramento è dovuto alla capacità del FC di anticipare il fenomeno del backtrack e quindi ridurre preventivamente lo spazio di ricerca. L'aggiunta del FC ad un algoritmo di BT è quindi in generale una buona idea.

Algoritmo 3 FORWARD CHECKING

Precondizioni: variabili X ($|X| = n$) con domini D e vincoli associati R, assegnazione corrente S, indice i della variabile corrente

Postcondizioni: se $true$, S consistente; se $false$, S è inconsistente

```
1: function FC(X, D, R, S, i)
2:     for all j : c = ⟨x_i, x_j⟩ ∈ R_i do        /* x_j vincolate a x_i tramite c */
3:         U ← {d ∈ D_j : ⟨s_i, d⟩ ∉ rel(c)}      /* valori inconsistenti di D_j */
4:         D_j ← D_j \ U
5:         if D_j = ∅ then
6:             return false                        /* assegnazione inconsistente */
7:         end if
8:     end for
9:     return true                                 /* assegnazione consistente */
10: end function
```

2.3.5 Arc consistency (AC)

Il meccanismo di constraint propagation descritto nel FC viene ampliato quando si comincia a parlare di **arc consistency**. Con questa accezione ci riferiamo ad una proprietà dei CSP alla quale è possibile ricondursi operando sul grafo delle variabili e dei vincoli, senza per questo alterare il significato del problema iniziale. Come è usuale nella letteratura relativa, parleremo di AC nel caso di CSP binari (e quindi di grafi in senso stretto). In particolare, è importante che il grafo dei vincoli sia reso orientato, ma per fare questo è sufficiente sostituire tutti gli archi non orientati con coppie di archi orientati $\langle x_i, x_j \rangle$ e $\langle x_j, x_i \rangle$.

Spieghiamo però cosa si intende per AC. Abbiamo già definito che cos'è la node consistency parlando di vincoli unari (2.2): l'AC è concettualmente analoga, ma stavolta in ambito di vincoli binari. La ragione della parola "arc" è chiara se si osserva che nel grafo del CSP i vincoli binari sono rappresentati proprio dagli archi. Rendere D_i **arc consistent** rispetto ad un arco orientato $\langle x_i, x_j \rangle$ del grafo dei vincoli significa eliminare tutti gli elementi di D_i che non sono in relazione con x_j, cioè che non soddisfano $\langle x_i, x_j \rangle$ per nessun valore di D_j. Rendere un CSP arc consistent equivale a fare quanto detto per ogni D_i del problema e rispetto ad ogni arco uscente da x_i.

Supponiamo di avere un CSP di sole 2 variabili x_1, x_2 con domini $D_1 = \{4, 10, 5, 6, 11\}$ e $D_2 = \{2, 7, 5, 0, 1\}$, ed una coppia di vincoli $c_1 = \langle x_1, x_2 \rangle$ e $c_2 = \langle x_2, x_1 \rangle$ descrivibile come "$x_1 + x_2 = 10$". Rendiamo il problema arc consistent. Gli elementi $\{4, 6, 11\}$ di D_1 non soddisfano c_1 per nessun valore

Algoritmo 4 AC-3

Precondizioni: variabili X ($|X| = n$) con domini D, vincoli C

Postcondizioni: se $true$, D_i arc consistent per $i = 1, 2, \ldots, n$; se $false$, problema inammissibile

```
1:  function AC3(X, D, C)
2:      Q ← C
3:      repeat
4:          c ← un qualunque arco ⟨x_i, x_j⟩ ∈ Q
5:          Q ← Q \ {c}                          /* elimina l'arco dalla coda */
6:          if REDUCE(c, D_i, D_j) then          /* rende D_i arc consistent */
7:              if D_i = ∅ then
8:                  return false                 /* problema inammissibile */
9:              else      /* D_i ridotto, D_k potenzialmente non arc consistent */
10:                 Q ← Q ∪ {⟨x_k, x_j⟩ : k ≠ j}          /* k = 1, 2, ..., m */
11:             end if
12:         end if
13:     until Q = ∅
14:     return true                              /* riduzione completata */
15: end function

16: function REDUCE(c, D_i, D_j)                 /* c = ⟨x_i, x_j⟩ */
17:     REDUCED ← false
18:     for all d_i ∈ D_i do
19:         S ← {d_j ∈ D_j : c({d_i, d_j}) = 1}
20:         if S = ∅ then
21:             D_i ← D_i \ {d_i}
22:             REDUCED ← true                   /* dominio modificato */
23:         end if
24:     end for
25:     return REDUCED
26: end function
```

di D_2, quindi $D_1^{'} = \{10, 5\}$; analogamente, gli elementi $\{2, 7, 1\}$ di D_2 non soddisfano c_2 per nessun valore di D_1, quindi $D_2^{'} = \{5, 0\}$. I nuovi domini $D_1^{'}$ e $D_2^{'}$ sono arc consistent, e di conseguenza lo è anche il problema. Nonostante la riduzione effettuata, l'espressività del CSP non è stata alterata. Su larga scala, l'impatto computazionale della riduzione è anzi notevolmente vantaggioso.

Il più conosciuto algoritmo per il conseguimento dell'AC è senza dubbio **AC-3** (= Arc Consistency #3), formulato da Alan Mackworth nel 1977 [4]. Il numero 3 è motivato dal fatto che esistono altri algoritmi meno recenti (AC-1, AC-2) e più recenti (AC-4, AC-5, ...) per raggiungere lo stesso scopo. In linea di massima, AC-3 mantiene il migliore equilibrio fra efficienza e difficoltà di implementazione, caratteristiche che lo rendono il più adatto per fini didattici.

L'integrazione con il BT si realizza ristabilendo l'AC ogni qualvolta si assegni con successo una nuova variabile. Per questo motivo non va trascurata la complessità dell'algoritmo di AC adottato in fase di implementazione. Non scenderemo nel dettaglio del calcolo, ma è dimostrabile che AC-3 ha complessità computazionale $\mathcal{O}(md^3)$ rispetto al numero m degli archi e alla dimensione d del dominio più grande. Altri algoritmi comunemente utilizzati hanno complessità minore di AC-3, come AC-4 che ha costo di caso peggiore pari a $\mathcal{O}(md^2)$.

2.4 Osservazioni

Abbiamo sottolineato che il BT, che è la struttura portante di gran parte degli algoritmi attuali di risoluzione di CSP, ha complessità esponenziale. D'altra parte non è noto, ad oggi, un algoritmo *efficiente* (polinomiale) per la risoluzione di un generico CSP. In termini informatici, si dice che i CSP sono problemi *intrattabili*. Abbiamo anche fornito un esempio (2.3.1) di come il tempo di esecuzione di un algoritmo esponenziale cresca molto velocemente con la dimensione dell'input. Ferma restando però l'"inefficienza" del BT, sono state illustrate alcune semplici accortezze per velocizzare la ricerca di una soluzione.

Ci si chiede se si possa fare di meglio. La prima cosa che ci si potrebbe domandare è se l'ordine di instanziazione delle variabili possa influire o meno sul tempo di esecuzione del BT. Questo è generalmente vero. Anche l'idea di instanziare le variabili partendo da quelle più vincolate sembra buona, in quanto riduce lo spazio di ricerca già nelle fasi iniziali del BT. Le due metodologie illustrate sono classici esempi di **euristiche**[6]. Esse dimostrano come sia di vitale importanza sfruttare le informazioni fornite a due livelli: a livello statico quantificando parametri correlati alla natura del problema, a livello dinamico deducendo misure significative dallo stato del BT (qualità delle assegnazioni, dimensioni attuali dei domini etc.). Attraverso un'euristica è possibile conferire al BT il tocco di *smartness* che spesso —se non sempre— fa la differenza.

Altri algoritmi molto efficaci per la risoluzione di CSP sono quelli di **ricerca locale**, ma per la loro descrizione si rimanda ad altri testi.

[6] anche il BJ può essere considerato un'euristica per il BT

Capitolo 3

Modellazione del cruciverba

3.1 Il cruciverba come CSP

Dopo il capitolo introduttivo sui CSP, possiamo ben dire che la creazione di un cruciverba appartiene a tale classe di problemi. Ricordiamo che un software di *creazione* di cruciverba è un software che, dati in input uno schema ed un elenco di parole, è in grado di restituire come output lo schema riempito con parole appartenenti a tale elenco.

A seconda di come si vogliano scegliere variabili e vincoli, è possibile ricondursi a più modelli del CSP. A tal proposito, introdurremo alcune definizioni formali:

Definizione 3.1 (Alfabeto) *Si definisce* alfabeto *l'insieme* Σ *dei valori ammissibili per un carattere.*

Nell'implementazione del software di risoluzione descritto nel capitolo 4, l'alfabeto Σ di riferimento sarà quello ASCII (A–Z). L'assunzione verrà mantenuta da questo momento in poi.

Definizione 3.2 (Pattern e Parole) *Sia* ε *il carattere nullo. Un* pattern s *è una sequenza ordinata di caratteri* $s_1 s_2 \dots s_n$ *tale che* $s_i \in \Sigma \cup \{\varepsilon\}$ *per* $i = 1, 2, \dots, n$*. Se* $s_i \neq \varepsilon$ *per* $i = 1, 2, \dots, n$ *il pattern prende il nome di* parola*.*

Definizione 3.3 (Dizionario) *Dato un alfabeto* Σ*, si definisce* dizionario *un insieme* $\mathcal{D}$ *di parole costruite su* Σ*.*

Su un dizionario deve essere possibile ricercare parole anche non specificandone tutti i caratteri. È cioè di interesse poter effettuare una primitiva operazione di *pattern matching*. Per poter riutilizzare la definizione appena fornita di pattern,

basterà associare al carattere nullo ε il significato di "qualsiasi carattere", ed effettuare le ricerche sul dizionario tramite stringhe di questo tipo. Riassumendo, diremo che un pattern $s = s_1 s_2 \dots s_n$ ha delle corrispondenze (o *matching*) in $\mathcal{D}$ se e solo se:

$$\exists w = w_1 w_2 \dots w_n \in \mathcal{D} : w_i = s_i \qquad \forall s_i \neq \varepsilon$$

Per comodità, indicheremo la presenza di corrispondenze di s in $\mathcal{D}$ con la notazione insiemistica $s \in \mathcal{D}$.

Le seguenti definizioni sono invece legate alla struttura del cruciverba:

Definizione 3.4 (Schema) *Si definisce* schema *cruciverbale (o griglia) la matrice $\mathcal{G}_{mn}$ di m righe ed n colonne delle caselle del cruciverba. Un elemento g_{ij} con $i \in \{1, 2, \dots, m\}$ e $j \in \{1, 2, \dots, n\}$ vale:*

$$\begin{cases} 0 & \textit{se la casella } \langle i, j \rangle \textit{ è vuota} \\ 1 & \textit{se la casella } \langle i, j \rangle \textit{ è annerita} \\ c & \textit{altrimenti} \end{cases}$$

dove c è un carattere dell'alfabeto Σ fissato.

Definizione 3.5 (Definizione) *Si definisce* definizione *di uno schema cruciverbale una sequenza consecutiva, orizzontale o verticale, di 2 o più caselle non annerite. La lunghezza della definizione è determinata univocamente dal numero delle suddette caselle.*

Quindi, in questo capitolo, useremo il termine definizione per riferirci non tanto all'indizio fornito dal gioco per giungere alla parola da inserire, quanto alla parola stessa. D'altra parte la generazione della lista delle definizioni vere e proprie è completamente esterna al problema della creazione dello schema, che invece è l'oggetto di questo studio.

Definizione 3.6 (Incrocio) *Si definisce* incrocio *una casella non annerita dello schema cruciverbale condivisa da 2 definizioni.*

È evidente che, essendo il cruciverba bidimensionale e 2 le direzioni di inserimento, un incrocio non può essere condiviso da più di 2 definizioni. Inoltre 2 definizioni possono incrociarsi se e solo se una è orizzontale e l'altra è verticale, o viceversa.

Di seguito vengono proposti due modelli del problema del cruciverba: nel primo modello le variabili sono le lettere, mentre nel secondo modello le variabili sono le definizioni. Entrambi i modelli effettuano le verifiche di consistenza su un alfabeto Σ ed un dizionario $\mathcal{D}$ prefissati.

3.2 Modello letter-based

Nel modello letter-based, come già detto, le variabili sono le lettere da inserire in ogni casella non annerita dello schema. Tali variabili possono essere assegnate purché i loro valori appartengano all'alfabeto Σ e formino parole presenti nel dizionario $\mathcal{D}$. In altri termini, l'ultima affermazione equivale a dire che le definizioni dello schema sono i vincoli del problema. Chiaramente, l'arità di un vincolo di definizione è pari alla lunghezza della definizione stessa. Riassumendo, il CSP letter-based è costituito da:

- un insieme di variabili $L = \{l_1, l_2, \ldots, l_n\}$ con domini iniziali $D_i = \Sigma$ corrispondenti alle caselle non annerite dello schema
- un insieme di vincoli $P = \{p_1, p_2, \ldots, p_m\}$ di arità $r_j \leq n$ che costituiscono le definizioni dello schema e coinvolgono sottoinsiemi $M_j \subseteq L$

Resta da chiarire la verifica di consistenza di un vincolo di definizione. A tale scopo prenderemo in considerazione l'assegnazione corrente di L:

Definizione 3.7 (Pattern associato) *Sia $V = \{v_1, v_2, \ldots, v_n\}$ l'assegnazione corrente di L, con $v_i = \varepsilon$ per ogni variabile non assegnata; sia p un qualsiasi vincolo ed $M = \{l_1, l_2, \ldots, l_r\}$ con $r \leq n$ il suo ambito associato. Chiameremo* pattern associato *a p tramite V ed indicheremo con* $\text{patt}(p, V)$ *il pattern formato dalle lettere $v_1 v_2 \ldots v_r$.*

Con l'aiuto di questa definizione, possiamo dire che p è soddisfatto da un'assegnazione V se e solo se il suo pattern associato rileva corrispondenze nel dizionario, cioè:

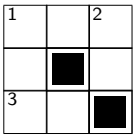
$$p(V) = 1 \iff \text{patt}(p, V) \in \mathcal{D}$$

È opportuno aggiungere un ulteriore vincolo, cioè quello di unicità delle definizioni. Siano p_{j_1} e p_{j_2} due qualsiasi vincoli distinti del problema, quindi $j_1, j_2 \in \{1, 2, \ldots, m\}$ con $j_1 \neq j_2$, e si supponga che entrambi i loro ambiti M_{j_1} ed M_{j_2} siano costituiti da tutte e sole variabili assegnate, cioè i loro pattern associati siano parole. Estendendo la scelta di j_1 e j_2 a tutte le possibili coppie di vincoli definite come sopra, il vincolo di unicità delle definizioni si esplicita formalmente in questo modo:

$$\text{patt}(p_{j_1}, V) \neq \text{patt}(p_{j_2}, V)$$

dove V continua ad essere l'assegnazione corrente.

Modelliamo a titolo esemplificativo il seguente cruciverba:



Le variabili (lettere) sono:

$$\begin{cases} l_1 \leftrightarrow g_{11} \\ l_2 \leftrightarrow g_{12} \\ l_3 \leftrightarrow g_{13} \\ l_4 \leftrightarrow g_{21} \\ l_5 \leftrightarrow g_{23} \\ l_6 \leftrightarrow g_{31} \\ l_7 \leftrightarrow g_{32} \end{cases}$$

dove $g_{ij} \neq 1$ con $g_{ij} \in \mathcal{G}$ per ogni i, j (caselle non annerite). Con la notazione $l \leftrightarrow g_{ij}$ intendiamo dire che la lettera l è associata al valore della casella alla posizione $\langle i, j \rangle$ dello schema cruciverbale. I vincoli (definizioni) sono:

$$\begin{cases} p_1 = \text{“1 ORIZZONTALE”} & \text{con } M_1 = \{l_1, l_2, l_3\} \\ p_2 = \text{“1 VERTICALE”} & \text{con } M_2 = \{l_1, l_4, l_6\} \\ p_3 = \text{“2 VERTICALE”} & \text{con } M_3 = \{l_3, l_5\} \\ p_4 = \text{“3 ORIZZONTALE”} & \text{con } M_4 = \{l_6, l_7\} \end{cases}$$

3.3 Modello word-based

Nel modello word-based, vengono considerate come variabili le definizioni dello schema. I vincoli sono tutti binari e corrispondono agli incroci, cioè le lettere condivise da più di una definizione. In altre parole, se due definizioni si incrociano, devono contenere la stessa lettera nella posizione in cui esse si incrociano. Il CSP è così formulato:

- un insieme di variabili $P = \{p_1, p_2, \ldots, p_n\}$ con domini iniziali $D_i = \mathcal{D}$ corrispondenti alle definizioni dello schema

- un insieme di vincoli binari $C = \{c_1, c_2, \ldots, c_m\}$ che rappresentano gli incroci fra le definizioni

Per assicurarsi che un incrocio sia rispettato, occorre ampliare il significato di vincolo in questo modello. Indicheremo con la notazione $c = \langle \langle p_{i_1}, h \rangle, \langle p_{i_2}, k \rangle \rangle$ il vincolo di incrocio fra due definizioni p_{i_1} e p_{i_2} (con $i_1, i_2 \in \{1, 2, \ldots, n\}$ ed $i_1 \neq i_2$) alle rispettive posizioni h e k; siano poi $p_{i_1}[h]$ la h-esima lettera di p_{i_1} e $p_{i_2}[k]$ la k-esima lettera di p_{i_2}, e si consideri che esse possono risultare nulle,

cioè pari ad ε. Il vincolo c è soddisfatto se e solo se:

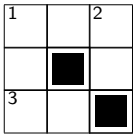
$$p_{i_1}[h] = p_{i_2}[k] \quad \text{oppure} \quad p_{i_1}[h] = \varepsilon \quad \text{oppure} \quad p_{i_2}[k] = \varepsilon$$

Appare evidente che la verifica della presenza delle definizioni nel dizionario $\mathcal{D}$ diventa nel modello word-based un vincolo unario, ovvero un semplice vincolo di dominio. Anche i controlli sulla correttezza lessicale (cioè i vincoli di alfabeto) scompaiono in quanto incapsulati nel significato stesso di dizionario. La maggiore comprensibilità di questo modello è dovuta al fatto che l'inserimento per definizioni è quello che risulta più vicino all'approccio umano.

Per finire, la verifica di unicità è immediata. Siano p_{i_1} e p_{i_2} due qualsiasi variabili distinte ed assegnate del problema, quindi $p_{i_1} \neq \varepsilon$ e $p_{i_2} \neq \varepsilon$ con $i_1, i_2 \in \{1, 2, \ldots, n\}$ e $i_1 \neq i_2$. Per tutte le coppie di variabili così definite deve essere:

$$p_{i_1} \neq p_{i_2}$$

Modelliamo ora lo stesso cruciverba utilizzato nell'esempio letter-based:



Le variabili (definizioni) sono:

$$\begin{cases} p_1 = \text{“1 ORIZZONTALE”} \\ p_2 = \text{“1 VERTICALE”} \\ p_3 = \text{“2 VERTICALE”} \\ p_4 = \text{“3 ORIZZONTALE”} \end{cases}$$

I vincoli (incroci) sono:

$$\begin{cases} c_1 = \langle\langle p_1, 1\rangle, \langle p_2, 1\rangle\rangle \\ c_2 = \langle\langle p_1, 3\rangle, \langle p_3, 1\rangle\rangle \\ c_3 = \langle\langle p_2, 3\rangle, \langle p_4, 1\rangle\rangle \end{cases}$$

cioè $p_1[1] = p_2[1]$, $p_1[3] = p_3[1]$ e $p_2[3] = p_4[1]$.

3.4 Confronto dei modelli

La scelta di uno o dell'altro modello dipende particolarmente dal tipo di algoritmo risolutivo che si intende implementare. La prima differenza sostanziale scaturisce dalla scelta dei domini. Nel modello letter-based i domini hanno tutti cardinalità costante e pari a $|\Sigma|$ che, nel caso del charset ASCII, è 26; nel

modello word-based i domini hanno invece cardinalità pari a $|\mathcal{D}|$, che è funzione del dizionario ricevuto in input. Una piccola semplificazione può essere operata indicizzando il dizionario per lunghezza delle parole, in modo da poter restringere il dominio iniziale di una definizione alle sole parole del dizionario della stessa lunghezza. Questa scelta appare ragionevole, ma in un dizionario di circa 150000 termini anche l'insieme più ridotto di parole di stessa lunghezza rimane nell'ordine delle migliaia, che è notevolmente maggiore di 26. A fronte di ciò, gli algoritmi strettamente legati ai domini (come FC e AC-3) risultano molto costosi nel caso si decida di adottare il modello word-based. Una scelta intelligente delle strutture dati per la memorizzazione dei domini potrebbe tuttavia ovviare a questo inconveniente.

D'altra parte, lo svantaggio del modello letter-based è quello di essere meno intuitivo dal punto di vista umano. L'enigmista di certo non crea il proprio cruciverba lettera per lettera. Si osservi inoltre che nella maggior parte dei casi il numero di variabili è più alto che nel modello word-based. Se però ci si pone in un'ottica prettamente informatica, è chiaro che il genere di variabili trattate sono diverse. Nel modello letter-based una variabile è un solo carattere; nel modello word-based è una stringa, quindi un vettore di caratteri. Non è perciò molto significativo un confronto quantitativo sul numero di variabili del problema.

Un'altra differenza risiede, come già rilevato, nell'arità dei vincoli dei due modelli: quello letter-based è costituito da vincoli n-ari con n non fissato, mentre quello word-based è strettamente identificabile come CSP binario. Questa particolarità ad esempio agevola la trascrizione in linguaggi di programmazione o software specifici per la risoluzione automatica di CSP, che in genere richiedono che i vincoli siano tutti binari.

Capitolo 4

Implementazione

4.1 Considerazioni generali

Questo studio si propone di realizzare un software per la *creazione* (compilazione) di cruciverba. I capitoli precedenti, tramite un percorso *bottom-up*, hanno introdotto concetti teorici che ponessero basi solide per il progetto. Nel capitolo 3 sono stati descritti poi due modelli intuitivi per la descrizione del CSP di creazione. Il compilatore realizzato adotta il modello letter-based, anche se con poche modifiche strutturali si potrebbero aggiungere le funzionalità word-based. Questo è possibile perché la struttura scelta per il dizionario —vitale per l'efficienza della compilazione— è già compatibile con entrambi i modelli. La generazione delle definizioni è stata infine tralasciata per i seguenti motivi:

- la scelta delle definizioni non interviene in alcun modo nel problema della compilazione del cruciverba

- automatizzare questa operazione richiederebbe la scrittura di un database di definizioni per ogni dizionario

- considerando che un dizionario contiene mediamente dai 200000 termini in su e che, ragionevolmente, andrebbero scritte almeno 3 definizioni per ogni termine, bisognerebbe scrivere un database di centinaia di migliaia di definizioni per ogni dizionario utilizzato

Il compilatore di cruciverba *crucio* è realizzato interamente in linguaggio ANSI C++. Non sono necessarie librerie esterne perché il programma si affida unicamente alla *Standard Template Library* C++ (STL).

Il software include anche il programma *cruciotex* (qui non documentato) per la generazione dell'output in formato LaTeX. Per la compilazione dell'output è necessario che nel sistema LaTeX locale sia installato il pacchetto `crossword` [13].

La lunghezza di alcune parti di codice non deve trarre in inganno in termini di tempo di esecuzione. L'apparente complessità è dovuta ad un alto numero di dichiarazioni, usate invece proprio per non pregiudicare la leggibilità dei sorgenti. Inoltre, il codice è stato scritto in modo da agevolare l'ottimizzazione da parte del compilatore. Gran parte dei metodi —si osservi in particolare la classe `Model`— sono infatti dichiarati `inline`. Il compilatore `gcc` [12] ha dato risultati molto positivi: la velocità di *crucio* è particolarmente differente a seconda che esso sia compilato normalmente o con tutte le ottimizzazioni (`gcc -O3`).

4.2 Struttura di *crucio*

Di seguito vengono descritti i sorgenti che costituiscono il cuore del compilatore. I moduli concettuali sono 6: dizionario, schema, modello, compilatore, output e utente.

4.2.1 Dizionario

È già stata accennata nel capitolo 3 la funzionalità di base che un dizionario deve fornire: il *pattern matching*. Il dizionario è stato concepito con questo obiettivo, quello di permettere la ricerca di parole tramite un *pattern*, cioè non specificandone tutte le lettere. Questo è indispensabile perché nel corso della compilazione del cruciverba vengono a costituirsi parole "parziali" delle quali è indispensabile verificare le corrispondenze nel dizionario. Tale situazione si riscontra sia nella compilazione letter-based che word-based, quindi il dizionario è adatto ad entrambi i modelli senza sostanziali modifiche.

File inclusi:

`Dictionary.h`, `Dictionary.cc`

Tipo `ABMask`

Il tipo `ABMask` è un alias per `bitset<26>`, dove `bitset` è una classe template della STL. Con un oggetto `ABMask` è possibile rappresentare in modo compatto un insieme di lettere dell'alfabeto A–Z, dove un bit della maschera è biunivocamente associato ad una lettera tramite la sua posizione nella maschera (da 0 a 25 inclusi). Un bit sarà pari ad 1 se la lettera identificata dalla sua posizione

è presente nell'insieme, e 0 se non è presente. Quindi ad esempio la maschera 00000000000100000101001000 equivale all'insieme {D, G, I, O}.

Classe `WordSet`

Un oggetto `WordSet` racchiude un sottoinsieme di parole di una stessa lunghezza `m_length`, e include le strutture di base per effettuare il pattern matching. In questa classe non vengono effettuati controlli di validità sulle parole inserite e si assume che le parole vengano aggiunte in ordine alfabetico. Quest'ultima specifica consentirà di velocizzare enormemente il pattern matching grazie al classico algoritmo di ricerca binaria. Tutti i requisiti di validità sull'input vengono comunque garantiti dalla classe `Dictionary`.

Ad ogni inserimento, la parola viene aggiunta alla fine di un vettore:

```
vector<string> m_words;
```

L'ID (o chiave) della parola nel `WordSet` è semplicemente l'indice all'interno di questo vettore, quindi parole lessicograficamente crescenti avranno ID crescenti. Parallelamente, viene mantenuta una hash table:

```
vector<vector<uint> > m_cpMatrix;
```

che mappa ogni coppia `<pos, ch>` alla lista delle parole che contengono il carattere `ch` alla posizione `pos`. In realtà queste liste non contengono le parole vere e proprie ma i loro ID, cioè gli indici in `m_words`. Attraverso il metodo:

```
const vector<uint>* getCPVector(const uint pos,
        const char ch) const;
```

è possibile ottenere le liste di parole appena descritte. Infine il metodo:

```
void getPossibleAt(const uint pos,
        ABMask* const possible) const;
```

salva in `possible` l'insieme di tutti i caratteri che nel `WordSet` appaiono alla posizione `pos`.

Classe `WordSetIndex`

Non è altro che una classe di supporto per incapsulare un insieme di `WordSet` con lunghezze incluse in un intervallo `[m_minLength, m_maxLength]`. Fondamentalmente `WordSetIndex` si comporta come un vettore di `WordSet`.

Classe `Dictionary`

La classe `Dictionary` incapsula un `WordSetIndex` ed implementa il pattern matching. Il dizionario viene di norma caricato da un file di input contenente una lista di parole (una per riga) in alfabeto ASCII (A–Z). Le parole che non osservano questa restrizione vengono scartate; dopo averne rimosso i duplicati, le parole valide vengono ordinate, convertite in maiuscolo ed inserite nei rispettivi `WordSet` in base alla lunghezza.

Il pattern matching è realizzato dal metodo:

```
bool getMatchings(const string& pattern,
        MatchingResult* const res,
        const set<uint>* const excluded = 0) const;
```

dove `MatchingResult` è un tipo "opaco" per il salvataggio del risultato, ed `excluded` (di default `NULL`) è l'eventuale insieme degli ID delle parole che si intendono escludere dal matching. È necessario richiedere al dizionario un oggetto `MatchingResult` con il metodo `createMatchingResult`; la successiva deallocazione va effettuata con il metodo `destroyMatchingResult`. L'algoritmo compie i seguenti passi:

1. seleziona il `WordSet` associato alla lunghezza di `pattern`
2. per ogni carattere non vuoto[1] di `pattern` ottiene la lista delle occorrenze tramite `getCPVector`
3. calcola le corrispondenze di `pattern` come l'intersezione fra tutte le liste ottenute al punto precedente
4. se `(excluded != NULL)`, dalle corrispondenze vengono rimossi gli ID in esso contenuti
5. salva la lista delle corrispondenze in `res`

Se il pattern è ad esempio "P?RT?", si cercano nel `wordset` di lunghezza 5 le corrispondenze di "P????", "??R??" e "???T?"; verosimilmente, questi pattern potrebbero rilevare rispettivamente {PALCO, PARTE, PENNA, PONTE, PORTA}, {CARLO, MERLO, PARTE, PORTA, SERIO} e {MONTE, PARTE, PORTA, SANTO}. L'intersezione delle corrispondenze è l'insieme {PARTE, PORTA}, i cui elementi rispettano effettivamente il pattern "P?RT?".

I metodi `getPossible` sono utili nell'implementazione letter-based perché, a seguito di un `getMatchings`, restituiscono le lettere con cui è possibile completare un pattern per giungere ad una parola del dizionario. Nel caso precedente,

[1] il carattere vuoto è un valore ASCII diverso da A–Z stabilito in `ANY_CHAR`

dal pattern "P?RT?" e dal risultato {PARTE, PORTA} è possibile dedurre che alla seconda posizione del pattern è possibile inserire le lettere {A, O}, mentre alla quinta posizione è possibile inserire le lettere {A, E}.

4.2.2 Schema

Di questo modulo fanno parte le classi che descrivono uno schema cruciverbale (o griglia).

File inclusi:

`Grid.h`, `Grid.cc`

Classe `Cell`

La classe annidata `Cell` amplia il significato di casella e lo lega al concetto di schema cruciverbale. Ogni casella contiene un riferimento `m_parent` allo schema a cui appartiene e le coordinate `m_row` e `m_column` al suo interno. È inoltre possibile:

- sapere se la casella è vuota (`isWhite`)
- sapere se la casella è annerita (`isBlack`)
- sapere se la casella è fissata (`isFixed`)
- sapere direttamente il valore della casella (`getValue`)
- sapere se la casella è un incrocio (`isCrossing`)
- ottenere le definizioni che partono dalla casella (`getAcrossDef` in orizzontale e `getDownDef` in verticale)
- muoversi verso le caselle adiacenti (`getNorth` in alto, `getWest` a sinistra, `getSouth` in basso, `getEast` a destra)

Con il metodo `getNearCells` si ottiene in modo rapido la lista delle caselle adiacenti non annerite.

Classe `Definition`

Un oggetto della classe `Definition` comprende tutte le informazioni sulla parola di uno schema, vale a dire:

- lo schema che la contiene (`getParent`)
- la casella iniziale (`getStartCell`)

- l'indice all'interno di uno dei due vettori di definizioni in `Grid`, a seconda della direzione (`getIndex`)
- la direzione, `ACROSS` se orizzontale o `DOWN` se verticale (`getDirection`)
- il numero della casella iniziale che localizza la definizione (`getNumber`)
- la lunghezza (`getLength`)

Con il metodo `getCrossingDefinitions` si ottiene la lista delle definizioni con le quali quella corrente si incrocia nello schema. La lista contiene coppie del tipo `<pos, <def, dPos>>` dove:

- `pos` è la posizione dell'incrocio nella definizione corrente
- `def` è il riferimento alla definizione incrociata
- `dPos` è la posizione dell'incrocio nella definizione incrociata

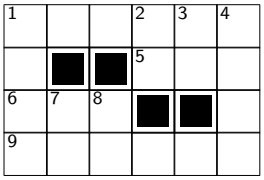
Classe `Grid`

La classe `Grid` fornisce un costruttore per caricare uno schema da file. Lo schema è immutabile, infatti i metodi di `Grid` sono tutti `const`. È nel costruttore di questa classe che vengono allocati i vari oggetti `Cell` e `Definition`, i cui costruttori sono però privati. Per questo motivo, in entrambe le classi `Grid` è dichiarata `friend`. In questo modo si garantisce che solo alla classe `Grid` sia concesso di instanziare caselle e definizioni.

A titolo esemplificativo, il file di input:

```
4
6
------
-##---
---##-
------
```

definisce uno schema di 4 righe e 6 colonne e corrisponde graficamente al seguente cruciverba:



Quindi nel file il simbolo `'-'` indica una casella vuota, mentre il simbolo `'#'` rappresenta una casella annerita. È possibile tuttavia fissare lettere A–Z nello schema, purché nel file siano inserite in maiuscolo[2].

A partire dal file di input, viene allocata dinamicamente una matrice di caselle:

```
vector<vector<Cell*> > m_cells;
```

di dimensione `m_rows * m_columns`, che nel caso illustrato valgono rispettivamente 4 e 6. Durante il caricamento vengono calcolati e salvati altri parametri significativi dello schema:

- il numero di caselle vuote (`getWhiteCells`)
- il numero di caselle annerite (`getBlackCells`)
- il numero di caselle fissate, cioè in A–Z (`getFixedCells`)
- il numero di parole (`getWords`)
- il numero di incroci (`getCrossings`)

Attraverso il metodo:

```
const Cell* getCell(const uint i, const uint j) const;
```

è possibile raggiungere le caselle `<i, j>` dello schema con le relative proprietà, fondamentali per la successiva creazione del modello.

4.2.3 Modello

Le classi seguenti contribuiscono alla costruzione del modello cruciverbale in termini di lettere e parole. Tramite l'input —dizionario e schema— il modello crea le fondamenta del CSP della compilazione.

File inclusi:

`Model.h`, `Model.cc`

`Letter.h`

`Word.h`

Classe `Letter`

La classe `Letter` rappresenta una lettera del modello e contiene un riferimento `m_cellRef` all'oggetto `Cell` associato nello schema. Con i metodi `get`, `set` e `unset` è possibile leggere e modificare la lettera.

[2] il requisito è chiaramente non restrittivo dal momento che un cruciverba è case-insensitive

Classe `Word`

La classe `Word` rappresenta una parola (parziale o completa) del modello e contiene un riferimento `m_defRef` all'oggetto `Definition` associato nello schema. Nel costruttore va fornito il dizionario di riferimento della parola, salvato poi nella variabile `m_dictionary`. Al suo interno è mantenuta inoltre una stringa `m_mask` che rappresenta il pattern associato alla parola. Con i metodi `get*`, `set*` e `unset*` è possibile leggere e modificare il pattern associato.

Il metodo fondamentale della classe è:

```
void doMatch(const bool updateLetters = false);
```

ed effettua internamente il pattern matching di `m_mask` su `m_dictionary`. Inoltre se `updateLetters` vale `true`, vengono salvati nel vettore `m_letterMasks` gli insiemi delle lettere possibili per il completamento del pattern. Tali insiemi sono calcolati internamente con il metodo `Dictionary::getPossible` e possono essere recuperati con i metodi `getAllowed`, usati ad esempio nel corso della compilazione letter-based per le verifiche di consistenza dei vincoli.

Classe `Model`

Il costruttore della classe `Model` è il seguente:

```
Model(const Dictionary* const d, const Grid* const g);
```

e si occupa di costruire gli oggetti necessaria per la formulazione sia del CSP letter-based che word-based. I vettori inizializzati per primi sono:

```
vector<Letter*> m_letters;
vector<Word*> m_words;
```

associati alle lettere (caselle vuote) e alle parole dello schema `g` e di rispettive dimensioni `g->getNonBlackCells()` e `g->getWords()`. Fatto ciò, viene allocata una matrice `m_mappings` di oggetti `Model::LWInfo` delle stesse dimensioni dello schema, i cui elementi sono strutture aggregate di questo tipo:

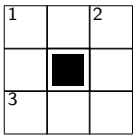
```
class LWInfo {
public:
    int m_li;
    int m_wiAcross;
    int m_wiDown;
    ...
};
```

La matrice descrive come lettere e parole si collocano nello schema. Se la casella i, j è vuota, l'elemento `m_mappings[i][j]` conterrà:

- nel campo `m_li` l'indice della lettera associata alla casella
- nel campo `m_wiAcross` l'indice della parola orizzontale che la contiene (se esiste, altrimenti -1)
- nel campo `m_wiDown` l'indice della parola verticale che la contiene (se esiste, altrimenti -1)

Le lettere vengono numerate da sinistra verso destra e dall'alto verso il basso; le parole vengono numerate nell'ordine delle lettere e, se dalla stessa casella partono due parole, quella orizzontale viene numerata prima di quella verticale. Questi numeri sono esattamente gli indici nei vettori `m_letters` e `m_words`.

Analizziamo ad esempio il seguente schema:



Lo schema è di dimensioni 3x3, le caselle vuote sono 8 e le parole sono 4, quindi vengono allocati:

- un vettore `m_letters` di 8 elementi
- un vettore `m_words` di 4 elementi
- una matrice `m_mappings` di tipo `lw_info[3][3]` con tutti i campi inizializzati a -1

Vengono assegnati i seguenti valori ai campi `m_li` di `m_mappings`:

$$M = \begin{pmatrix} 0 & 1 & 2 \\ 3 & -1 & 4 \\ 5 & 6 & 7 \end{pmatrix}$$

dove $M_{ij} = -1$ identifica una casella i, j annerita. Le parole dello schema sono nell'ordine: "1 ORIZZONTALE", "1 VERTICALE", "2 VERTICALE" e "3 ORIZZONTALE". I campi `m_wiAcross` saranno:

$$M = \begin{pmatrix} 0 & 0 & 0 \\ -1 & -1 & -1 \\ 3 & 3 & 3 \end{pmatrix}$$

mentre i campi `m_wiDown` saranno:

$$M = \begin{pmatrix} 1 & -1 & 2 \\ 1 & -1 & 2 \\ 1 & -1 & 2 \end{pmatrix}$$

dove $M_{ij} = -1$ significa che nessuna parola (orizzontale in `m_wiAcross` e verticale in `m_wiDown`) passa per la casella i, j.

Dallo schema e dalla matrice `m_mappings`, il costruttore crea ulteriori vettori per rappresentare le relazioni che intercorrono fra gli elementi di `m_letters` e di `m_words`. Sostanzialmente, questi sono per le lettere:

- i domini iniziali (`getInitLettersDomains`)
- le parole a cui appartengono (`getLettersWords`)
- le lettere nelle stesse parole (`getLettersNeighbours`)

e per le parole:

- i domini iniziali (`getInitWordsDomains`)
- le lettere contenute (`getWordsLetters`)
- le parole incrociate (`getWordsNeighbours`)

Si osservi che in tutti i suddetti vettori ci si riferisce a lettere e parole tramite i loro indici in `m_letters` e `m_words`.

La struttura della classe `Model` diventa decisamente più comprensibile se si esegue il programma *crucio* in modalità verbose, come vedremo in seguito.

4.2.4 Compilatore

Il compilatore opera su un oggetto `Model` per la ricerca della prima soluzione possibile del CSP. Coerentemente con quanto detto nel capitolo 3, sono state realizzate due implementazioni, una per il modello letter-based (classe `LetterCompiler`) ed una per il modello word-based (classe `WordCompiler`). Gli algoritmi utilizzati sono BJ e FC. Il BJ utilizzato nelle implementazioni del compilatore è euristico e quindi non esaustivo, ma giunge ad una soluzione molto più in fretta che con un BT completo. Una componente di pseudocasualità è infine introdotta dall'uso della funzione `rand` della libreria C nella scelta dei valori dei domini durante il backtracking.

File inclusi:

`Walk.h`, `Walk.cc`

`Backjumper.h`, `Backjumper.cc`

`Compiler.h`, `Compiler.cc`

`LetterCompiler.h`, `LetterCompiler.cc`

`WordCompiler.h`, `WordCompiler.cc`

Classe `Walk`

Le classi che estendono la classe astratta `Walk` definiscono l'euristica di ordinamento (statico) delle variabili. Nel programma sono implementate due sottoclassi `BFSWalk` e `DFSWalk` che corrispondono alla visita in ampiezza (*breadth-first search* o BFS) ed in profondità (*depth-first search* o DFS) del grafo delle variabili. Per grafo delle variabili ci riferiamo a quello costruito partendo dallo schema, ovviamente diverso a seconda del modello adottato. In particolare:

- nel grafo letter-based i nodi sono le caselle non annerite; in ogni nodo le adiacenze sono le caselle non annerite circostanti (`Cell::getNearCells`)
- nel grafo word-based i nodi sono le definizioni; in ogni nodo le adiacenze sono le definizioni incrociate (`Definition::getCrossingDefinitions`)

Il nodo di partenza per la visita —sia BFS che DFS— è la prima casella o definizione a partire dall'angolo in alto a sinistra.

Una sottoclasse di `Walk` deve quindi realizzare l'algoritmo di visita per entrambi i grafi. I metodi da implementare sono:

```
void visitLetters(const Model& m,
        vector<uint>* const order) const = 0;
void visitWords(const Model& m,
        vector<uint>* const order) const = 0;
```

dove `m` è il modello complessivo del problema e `order` è il vettore di destinazione per contenere gli indici delle variabili (rispettivamente in `m.getLetters()` e `m.getWords()`) ordinate secondo l'algoritmo di visita. È assolutamente obbligatorio che il vettore `order` risultante abbia elementi non ripetuti e sia di dimensione `m.getLettersNum()` nel primo caso e `m.getWordsNum()` nel secondo caso: ciò equivale a dire che il `Walk` deve visitare tutte le variabili del problema.

Classe `Backjumper`

La classe `Backjumper` include la logica del BJ (2.3.3). In un oggetto `Backjumper` viene mantenuta una lista di "salti efficaci" da effettuare in caso di backtrack

per assicurarsi che l'algoritmo torni a reinstanziare una variabile da cui molto probabilmente il backtrack è dipeso. Le informazioni sulle quali l'oggetto si basa sono il vettore che rappresenta l'ordine di instanziazione delle variabili ed il vettore delle *dipendenze*. Questi dati vanno necessariamente forniti chiamando il metodo:

```
void configure(const vector<uint>& order,
        const vector<vector<uint> >& deps);
```

Il vettore `deps` deve essere di dimensione pari al numero di variabili (come anche `order`), e deve contenere alla posizione i-esima la lista degli indici delle variabili da cui l'i-esima variabile dipende. La richiesta di salto viene effettuata con il metodo:

```
void jump(const uint i,
        const set<uint>* const failed = 0);
```

dove `i` è la variabile che ha generato il backtrack e `failed` è un eventuale insieme di altre variabili che sono intervenute nel backtrack stesso. Le variabili vengono localizzate tramite il vettore `m_order` di ordinamento. La chiamata ha esito positivo se e solo se il successivo metodo `isExhausted` restituisce `false`, cioè i salti possibili non sono stati esauriti.

Nel caso in cui `jump` abbia successo, l'origine e la destinazione dell'ultimo salto richiesto si ottengono con i metodi `getOrigin` e `getDestination`. L'origine coinciderà con il parametro `i` dell'ultima chiamata, mentre la destinazione sarà il più alto *safe jump* possibile.

Classe `Compiler`

La classe base dei compilatori è `Compiler`. Le sottoclassi devono implementare tre callbacks:

```
virtual void configure(const Walk& w) = 0;
virtual void reset() = 0;
virtual bool compileFrom(const uint i) = 0;
```

In `configure` vanno configurati gli oggetti interni necessari per l'esecuzione dell'algoritmo attraverso il `m_model` corrente ed il `Walk` fornito come parametro; in `reset` va resettato lo stato dell'algoritmo per una nuova esecuzione; in `compileFrom` va avviato il backtracking, con `i` pari all'indice della variabile corrente nell'ordinamento ed inizialmente uguale a 0.

Le sottoclassi dovranno tenere necessariamente conto dei parametri di compilazione impostati con i seguenti metodi:

- `setUnique`, per impostare nel CSP il vincolo di unicità delle parole

- `setDeterministic`, per forzare la ricerca di una soluzione deterministica

- `setVerbose`, per stampare sullo standard output i passi dell'algoritmo

Il determinismo è un requisito dei crucintarsi, ed è esprimibile con il vincolo che ad ogni fase del gioco il solutore possa compiere l'inserimento di almeno una parola con assoluta certezza. La verifica di determinismo *inline* in fase di compilazione è molto complessa, quindi il compilatore si limita a riavviare il backtracking finché la soluzione trovata non soddisfa il vincolo di determinismo. Con questa semplificazione, è sufficiente effettuare il controllo sul determinismo a posteriori con il metodo privato `isDeterministicSolution`. Lo svantaggio di questa scelta è che non è possibile prevedere l'inesistenza di soluzioni deterministiche per lo schema di input: se ciò accade, il programma si ritrova infatti a ciclare indefinitamente. Questa evenienza è comunque rara e si riscontra solo nel caso di schemi molto vincolati, non certo nelle geometrie classiche dei crucintarsi.

Il metodo che avvia la compilazione è:

```
Result compile(Model* const m, const Walk& w);
```

con `Result` definito come:

```
enum Result {
    SUCCESS,
    FAILURE_IMPOSSIBLE,
    FAILURE_OVERCONSTRAINED,
    FAILURE_ND_GRID
};
```

e richiama internamente le tre callbacks definite all'inizio del paragrafo. Con il riferimento `m` viene inizializzata la variabile membro protetta `m_model`, mentre il parametro `w` è lo stesso che `configure` riceve in ingresso. Il risultato restituito sarà:

- `SUCCESS`, se è stata trovata una soluzione, costituita dagli elementi dei vettori `m->getLetters()` e `m->getWords()` subito dopo la fine della compilazione

- `FAILURE_IMPOSSIBLE`, se il backtracking è terminato senza giungere ad alcuna soluzione

- `FAILURE_OVERCONSTRAINED`, se almeno una variabile del CSP ha un dominio iniziale vuoto
- `FAILURE_ND_GRID`, se è stata richiesta una soluzione deterministica ma lo schema induce il non-determinismo

Si ricade nel quarto caso quando non c'è nessuna lunghezza l tale che l'insieme di parole di lunghezza l sia costituito da uno ed un solo elemento. La conseguenza di questa situazione è che per qualsiasi soluzione la prima parola di un crucintarsio non potrà mai essere inserita con certezza.

Come per la struttura del modello, per capire meglio il funzionamento del compilatore è consigliato eseguire *crucio* in modalità verbose.

4.2.5 Output

Questa parte del programma si occupa dell'output avanzato del programma.

File inclusi:

`Output.h`, `Output.cc`

Classe `Output`

La classe `Output` gestisce i risultati di *crucio*. Sono disponibili due costruttori:

```
Output(const Model& m);
Output(istream& in);
```

per leggere i risultati da un modello `m`[3] o da uno stream `in` prodotto da una precedente esecuzione di *crucio* (4.2.6). Una volta costruito l'oggetto, è possibile stamparne il contenuto con il metodo:

```
void printRaw(ostream& out) const;
```

in un formato descritto nell'appendice A. Con il metodo:

```
void printLatex(ostream& out,
        const bool solution = false,
        const bool fillIn = false) const;
```

è possibile stampare il risultato in formato LaTeX [13]. Il parametro `solution` indica se includere la soluzione dello schema; il parametro `fillIn` specifica se l'output va renderizzato come crucintarsio invece che come cruciverba. Nel caso dei cruciverba le definizioni, come già detto, non sono incluse. Nell'output LaTeX è tuttavia possibile inserirle manualmente prima della compilazione. Il metodo `printLatex` è utilizzato dal programma *cruciotex*.

[3] che si suppone risolto

4.2.6 Utente

Di questo modulo fa parte la sola funzione `main`, che è l'entry-point del programma.

File inclusi:

`crucio.h`, `crucio.cc`

Funzione `main`

La sintassi per la linea di comando è questa:

```
crucio [-f <letter|word>] [-w <bfs|dfs>] [-r <seed>]
       [-U] [-D] [-v]
       -d <dictionary> -g <grid> <output>
```

dove gli argomenti fra parentesi quadre sono opzionali. In particolare:

`-f <letter|word>` (default=`letter`), specifica la compilazione per lettere (`LetterCompiler`) o per parole (`WordCompiler`)

`-w <bfs|dfs>` (default=`bfs`), specifica l'ordine di instanziazione delle variabili secondo `BFSWalk` o `DFSWalk`

`-r <seed>` (default=funzione di `time(0)`), specifica il random seed per la funzione C `srand` per la generazione dei numeri pseudocasuali

`-U`, attiva il vincolo di unicità delle parole

`-D`, forza la ricerca di una soluzione deterministica

`-v`, imposta la modalità verbose, ovvero stampa sullo standard output informazioni dettagliate sulla creazione del modello e sull'algoritmo di compilazione

`-d <dictionary>`, specifica il dizionario di input

`-g <grid>`, specifica lo schema di input

`<output>`, specifica il file binario di output

Il file di output creato è gestito dalla classe `Output` ed il suo formato è descritto nell'appendice A.

Capitolo 5

Prestazioni

5.1 Applicazioni reali

Il comportamento di *crucio* è influenzato da tre fattori: la struttura del dizionario, la complessità dello schema ed il modello adottato.

Prima di tutto, affinché la compilazione termini in tempi ragionevoli, è opportuno che il dizionario contenga più parole possibili. Un dizionario di piccole dimensioni causerebbe con facilità un alto numero di inconsistenze con conseguenti backtrack. Un buon dizionario dovrebbe contenere almeno 200000 parole. Anche la qualità delle parole contenute è molto significativa: se un dizionario contiene parole "scadenti" (verbi coniugati, parole sconosciute etc.) non si otterrà mai un cruciverba accettabile. Purtroppo i dizionari reperiti in rete non sono stati in grado di soddisfare questo importante requisito.

Non tutti gli schemi sono compilabili agevolmente da *crucio*. Le geometrie statunitensi e britanniche danno di solito risultati in pochi secondi o al massimo minuti. I crucintarsi sono gli schemi più facili da compilare. Se si considera infatti che nei crucintarsi classici spesso si esclude la presenza di parole adiacenti, la quantità di vincoli del CSP associato è molto bassa. È per questo che *crucio* è più indicato per creare crucintarsi piuttosto che cruciverba. Schemi cruciverbali realistici con "piazze" (= aree bianche) molto estese mettono infatti in seria difficoltà la compilazione. D'altronde il costo del BT è esponenziale, quindi non ci si può aspettare molto di meglio. In pratica, un qualsiasi cruciverba della *Settimana Enigmistica* potrebbe richiedere tempi di compilazione eccessivamente lunghi.

Quanto alla scelta del modello, il compilatore letter-based si è rivelato più efficiente in termini sia di tempo di esecuzione che di memoria utilizzata. Uno

dei motivi di fondo è la gestione molto più efficiente dei domini rispetto al compilatore word-based. Delle due euristiche di ordinamento delle variabili —sempre nel caso letter-based— la BFS ha dato senza dubbio risultati più convincenti.

5.2 Confronto con *cwc*

Con l'aiuto di qualche modifica nei sorgenti, sono stati effettuati alcuni paragoni con *cwc* [6]. Per i benchmarks è stato utilizzato un PC portatile Dell C610 con processore Pentium III Mobile 1GHz e 256MB di SDRAM PC133.

Schema	Tempo di esecuzione con *cwc*	Tempo di esecuzione con *crucio*
01	00:00.78	00:01.40
02	NC	NC
03	NC	NC
04	00:00.85	00:01.21
05	00:00.97	00:04.25
06	00:00.87	00:01.28
07	NC	00:28.99
08	00:00.88	00:01.30
09	02:20.58	00:09.28
10	NC	NC
11	00:40.86	03:01.29
12	00:00.97	00:01.75
13	00:02.34	00:02.04
14	00:00.91	00:01.38
15	00:00.77	00:01.09
16	NC	00:01.70
17	02:24.63	00:28.72
18	NC	NC
19	00:57.57	NC
20	00:00.88	00:01.51
21	00:10.01	00:02.72
22	00:06.18	00:33.24
23	00:02.26	01.19.42
24	02:20.96	00:09.37
25	00:02.66	00:05.01

Tabella 5.1: Tempi di esecuzione di *cwc* e *crucio*

I dizionari confrontati condividono la stessa idea di base per implementare il pattern matching. Tuttavia, *crucio* ha dimostrato di essere più veloce in tutti i casi analizzati: il pattern matching è risultato anche 10 volte più veloce. Il compromesso è però una maggiore richiesta di memoria.

Non è stato possibile stabilire in assoluto quale dei due programmi giunga ad una soluzione dello stesso schema più in fretta. C'è da dire che, a differenza di *crucio*, *cwc* non supporta i controlli di unicità e determinismo.

Sono stati effettuati test su 25 schemi —allegati con il software— usando uno stesso dizionario. Dal momento che il compilatore di *cwc* è letter-based, *crucio* è stato testato in tale modalità. In particolare, *cwc* è stato eseguito con le opzioni "`-w flood`" e *crucio* con le opzioni "`-f letter -w bfs`". Infine sono state eliminate le componenti di pseudocasualità in entrambi i programmi, anche se le differenti euristiche di BJ adoperate dai compilatori hanno inevitabilmente portato a tempi di esecuzione e output diversi. I programmi hanno infatti esibito comportamenti decisamente eterogenei, come osservabile nella tabella 5.1. La notazione "NC" indica il superamento del tempo di esecuzione massimo fissato a 5 minuti.

Come lavoro futuro, sarebbe di sicuro più realistico compiere dei benchmarks includendo le componenti pseudocasuali per poter piuttosto stilare una sintesi dei tempi medi di compilazione. La casualità d'altronde può influenzare i tempi di esecuzione sia in positivo che in negativo, determinando di fatto la "fortuna" delle scelte operate dal BT. Non è raro che, se *crucio* rimane bloccato nella compilazione per molto tempo, arrestando e riavviando il programma con un random seed diverso si giunga ad una soluzione in pochi istanti.

Appendice A

Output di *crucio*

Il file binario generato dall'esecuzione di *crucio* ha la seguente struttura:

```
struct cro_t {
    uint32_t m_rows;        /* numero di righe dello
                             * schema */
    uint32_t m_columns;     /* numero di colonne dello
                             * schema */
    char* m_unfilled;       /* vettore di dimensione
                             * [m_rows * m_columns]
                             * contenente lo schema
                             * iniziale */
    char* m_filled;         /* vettore di dimensione
                             * [m_rows * m_columns]
                             * contenente lo schema
                             * completato */
    uint32_t m_defsNum;     /* numero di definizioni */
    struct def_t* m_defs;   /* vettore di dimensione
                             * [m_defsNum] contenente
                             * le definizioni */
};
```

con `def_t` definita come:

```
struct def_t {
    uint16_t m_direction;   /* 0 = orizzontale
                             * 1 = verticale */
    uint16_t m_number;      /* indice della casella
```

```
                                * di partenza */
        uint32_t m_row;         /* riga della casella
                                * di partenza */
        uint32_t m_column;      /* colonna della casella
                                * di partenza */
        uint32_t m_clueLength;  /* lunghezza della
                                * definizione */
        char* m_clue;           /* stringa di lunghezza
                                * [m_clueLength]
                                * contente la soluzione
                                * della definizione */
    };
```

In particolare, nei vettori `m_unfilled` e `m_filled` lo schema viene memorizzato in `m_rows` righe contigue di dimensione `m_columns`. Di conseguenza un elemento `<i, j>` è accessibile all'indice `i * m_columns + j`. Si noti che i campi numerici sono memorizzati in notazione big endian (network byte order).

Bibliografia

[1] S. Russell, P. Norvig: *Artificial Intelligence: A Modern Approach*, 2nd ed., Prentice Hall, 2006

[2] D. E. Knuth: *The Art of Computer Programming, Vol. III: Searching and Sorting*, Addison-Wesley, 1973

[3] Wikipedia: `http://en.wikipedia.org`

[4] A. K. Mackworth: *Consistency in Networks of Relations*, 1977

[5] S. C. Jensen: *Crossword Compilation Programs Using Sequential Approaches*, January 1997

[6] L. Christensen: *A CrossWord Compiler*, version 1.0.1, September 1999, `http://cwordc.sourceforge.net`

[7] M. L. Ginsberg, M. Frank, M. P. Halpin, M. C. Torrance: *Search Lessons Learned from Crossword Puzzles*, 1990

[8] L. J. Mazlack: *Computer Construction of Crossword Puzzles Using Precedence Relationships*, 1976

[9] J. Connor, J. Duchi, B. Lo: *Crossword Puzzles and Constraint Satisfaction*, 2005

[10] W. J. Teahan, I. H. Witten: *Storing and Retrieving Keys in a Table by Cross-Indexing*

[11] *WebCrow*, Facoltà di Ingegneria dell'Università degli Studi di Siena: `http://webcrow.dii.unisi.it`

[12] GCC, The GNU Compiler Collection, `http://gcc.gnu.org`

[13] A LaTeX Package for Typesetting Crossword Puzzles: `http://www.ctan.org/tex-archive/macros/latex/contrib/gene/crossword`