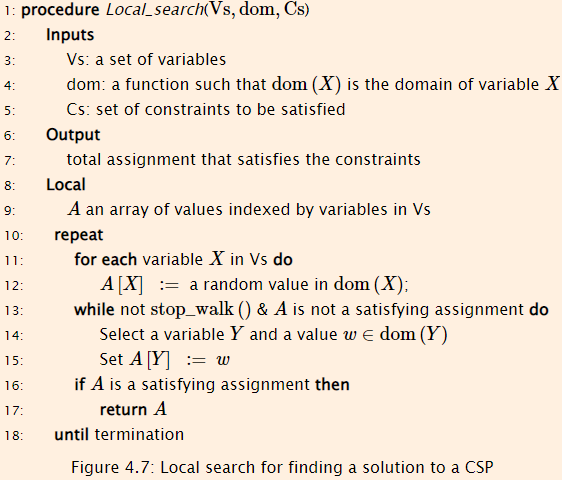
**RICERCA LOCALE E OTTIMIZZAZIONE**

Gli algoritmi precedenti ricercano sistematicamente lo spazio delle assegnazioni di valori alle variabili. Se lo spazio è finito, troveranno una soluzione o segnaleranno che non esiste alcuna soluzione. Il problema si presenta nel momento in cui lo spazio delle assegnazioni è infinito, in tali casi una ricerca sistematica dal punto di vista computazionale sarebbe troppo costosa e non ci assicura completezza, per cui per trattare questi problemi si usano algoritmi di ricerca locale che si focalizzano su zone della ricerca, anche questi non garantiscono di trovare una soluzione, ma permettono di affrontare il problema in un tempo ragionevole.

I metodi di ricerca locale iniziano con un'assegnazione totale di un valore a ciascuna variabile e cercano di migliorare questa assegnazione in modo iterativo eseguendo passaggi di miglioramento.

**LOCAL SEARCH ALGORITHM:**



L’array A usato dall’algoritmo tiene traccia degli assegnamenti fatti su ogni variabile che descrive il CSP. Il primo ciclo for\_each viene detto **random initialization,** si dà vita ad una assegnazione casuale che in seguito sarà migliorata, tutte le successive volte che verrà rieseguito questo ciclo ci troveremo difronte ad un **random restart** usato per risolvere alcuni problemi di convergenza ad una soluzione. In entrambi i casi sia di random initialization che di random restart questa assegnazione è una prova detta TRY perché appunto si va a provare che quella sia una soluzione e in seguito che possa diventarlo.

Il ciclo **while** è quello che si occupa della ricerca locale e per farlo genera dei successori dell’assegnazione casuale andandone a variare una singola variabile. La ricerca locale fatta da tale ciclo viene interrotta nel qual caso una assegnazione casuale viene trovata oppure nel momento in cui la funzione stop\_walk() sia soddisfatta per effettuare un random restart segno che nella zona dove si era focalizzata la ricerca locale non c’è nulla di buono.

Il **criterio di stop** risulta di vitale importanza per questo algoritmo perché lo può far mutare da un algoritmo completo ad uno incompleto.

**RANDOM SAMPLING**

Risulta una variante dell’algoritmo di local\_search, ove qui definiamo un criterio di stop che permette all’algoritmo di essere completo anche se poco efficiente in pratica **stop\_walk() è sempre a true e per cui il ciclo while non sarà mai eseguito**, né consegue che la ricerca locale non viene mai eseguita e l’algoritmo cicla effettuando ad ogni passo assegnamenti casuali che andrà a testare come soluzioni, queste soluzioni e questi assegnamenti casuali fanno a campionare lo spazio delle soluzioni in modo randomico permettendo all’algoritmo di esplorare diverse zone dello spazio di ricerca senza focalizzarsi su di una in particoare.

**RANDOM WALK**

In questa variante si ragione non eseguendo mai il passo di random restart per cui una volta avvenuta l’assegnazione iniziale l’algoritmo entrerà in un random walk dove andrà a migliorare l’assegnazione di partenza. **La stop\_walk() è posto sempre a false in questo caso e l’algoritmo termina quando trova una soluzione** per cui è completo, ma richiede un tempo eccessivo di calcolo. In questo caso con il random walk si esplora casualmente lo spazio circostante alla precedente assegnazione e questa è la differenza tra il random sampling e il random walk mentre il random sampling prova differenti zone dello spazio di ricerca in continuazione il random walk si focalizza su di una e si sposterà verso una nuova zona solo se nella precedente non ha trovato una soluzione.

**ITERATIVE BEST IMPROVEMENT**

Questo algoritmo risulta essere un local search algorithm dove la selezione dei successori di un assegnazione casuale viene fatta in base ad una funzione di valutazione che sceglierà il miglior successore tra tutti i possibili in particolare se il criterio è quello di massimizzare l’algoritmo prende il nome di **HILL CLIMBING** or **GREEDY ASCENT**, mentre se l’obbiettivo è minimizzare la funzione di scelta allora parliamo di **GREEDY DESCENT**. Una funzione di valutazione comune è quella che conta il numero di conflitti violati.

*Supponiamo che la discesa avida inizi con l'incarico* A=2*,* B=2*,* C=3*,*D=2*,* E=1*. Questa assegnazione ha una valutazione di 3, perché viola* A≠B*,* B≠D*,* C<D*. Un possibile successore con la valutazione minima ha* B=4 *con valutazione di* 1 *perché solo* C<D *è insoddisfatto. Questa assegnazione è selezionata. Questo è un minimo locale. Modificando il valore di D e impostandolo a 4 è possibile ottenere un possibile successore con il minor numero di conflitti , che ha una valutazione di 2(ovvero 2 conflitti non vietati). Può quindi cambiare A impostandolo a 4, con una valutazione di* 2*, ed infine cambiare*B impostandolo a 2*con una valutazione di 0, e si trova una soluzione.*

**A picture containing text

Description automatically generated**

L’iterative best improvement ad ogni passo considera il miglior successore tra quelli disponibili senza tenere in considerazione che il successore possa avere un valore di valutazione peggiore rispetto all’attuale assegnazione e questo causa all’algoritmo il problema dell’incompletezza dato da ottimi locali e non globali. Quindi resta intrappolato in minimi locali.

**RANDOMIZED ALGORITHMS [algoritmi randomizzati]**

Per affrontare il problema dell’incompletezza dell’iterative best improvement abbiamo 2 tecniche:

* Random restart (già visto nell’algoritmo di ricerca locale generale)
* Random walk (ovvero ai passi di miglioramento fatti in base alla funzione di valutazione vengono intermezzati passi di miglioramento eseguiti in maniera casuale come visto nella variante del local search ovvero il random walk.)

In particolare, quando i passi dell’iterative best improvement vengono mischiati a quelli casuali ci troviamo difronte ad una classe di algoritmi chiamati di **ricerca locale stocastica.**

Ogni qualvolta che utilizziamo **random walk in combinazione con il greedy descent in generale riusciamo a risolvere il problema dei minimi locali** tuttavia quale sia il migliore approccio per affrontare tali problemi e in particolare il loro spazio di ricerca è un problema empirico che solo la pratica e test svolti in modo differente possono snocciolare.

**LOCAL SEARCH VARIANTS: [variante randomiche all’approccio dato dal iterative best improvement]**

Per come abbiamo visto fino ad ora la ricerca locale non tiene traccia(memoria) dei passi fatti in precedenza (cosa che abbiamo visto nel IBI porta all’incompletezza). Un modo per migliorare la ricerca locale è introdurre degli stati di memoria e un algoritmo che fa questo è detto **TABU SEARCH**. L’idea del tabu search è quella di non riassegnare le variabili che sono state da poco selezionate per una riassegnazione. Questo limite tiene in conto un numero di passi, ad esempio **t** e solo le variabili che sono state cambiante prima degli ultimi **t** passi possono essere nuovamente riassegnate, l’intero **t** che tiene conto dei passi da tenere in memoria prendi il nome di **tabu tenure**  ed è un parametro che può essere ottimizzato**.**

Le varianti algoritmiche differiscono dalla computazione che essi richiedono per effettuare la scelta del miglior passo iterativo. In questa scelta computazionale ci sono 2 estremi ove da una parte si garantisce che il successore selezionato sia il migliore possibile dall’altra parte se la situazione di ricerca non produce risultati significativi si effettuano passi randomizzati per uscire da zone che risultano essere “cattive”(non hanno soluzioni).

Il tabu search grazie alla lista delle azioni effettuate ha memoria delle assegnazioni percorse e questa memoria gli permette di affrontare i plateau dello spazio di ricerca al fine di uscirne e superarli.

**MOST IMPROVING STEP**

Seleziona una delle coppie variabile-valore che produce il miglior miglioramento, anche se tale miglioramento è negativo. Questo algoritmo richiede una complessità di scelta ad ogni passo data da  dove n è il numero delle variabili, d è la cardinalità del dominio delle variabili e r è il numero di vincoli per ogni variabile.

Un'alternativa più sofisticata consiste nell'avere una coda di priorità di coppie variabile-valore con pesi associati. Per ogni variabile X, e ogni valore v nel dominio di X tale che X non è assegnato a v nell'attuale assegnazione, la coppia ⟨X, v⟩ sarebbe nella coda prioritaria.

Questo algoritmo impiega molto tempo a mantenere le strutture dei dati per garantire che ogni volta venga eseguito il passo più migliorativo

**TWO STAGE CHOICE [selezione variabile e poi selezione del valore]**

Un'alternativa consiste nel selezionare prima una variabile e quindi selezionare un valore per tale variabile. Per selezionare la variabile mantiene una coda di priorità di variabili, dove il peso di una variabile è il numero di conflitti a cui partecipa(la priorità aumenta all’aumentare dei conflitti). In seguito, il valore è assegnato minimizzando il numero di conflitti oppure assegnato in modo casuale. Una volta effettuata la riassegnazione vi è una fase di aggiornamento dei pesi per calcolare le priorità nella coda priorità che permette di selezionare la variabile con più conflitti.

La complessità per ogni passo risulta O(rk log(n)) dove n # di variabili, r # di vincoli e k # di variabili per il vincolo.

**ANY CONFLICT**

Qui la scelta si basa sul selezionare una variabile che partecipa a qualsiasi conflitto(questo permette una scelta della variabile conflittuale rispetto ai precedenti algoritmi con minor costo computazionale nella scelta). In seguito, si assegna un valore minimizzando i conflitti oppure in modo casuale.

Ci sono 2 varianti dell’any conflict:

* Si seleziona il conflitto(ovvero il vincolo) in maniera casuale e in seguito casualmente una variabile del conflitto. [diverse probabilità per una variabile di essere scelta]
* Si seleziona direttamente una variabile che sia conflittuale indipendentemente dal conflitto. [stessa probabilità per ogni variabile di essere scelta]

Per la scelta delle variabili conflittuali viene usata una struttura ad albero binario che ne faciliti il reperimento e l’aggiornamento.

**SIMULATED ANNEALING**

Questa variante non utilizza una struttura dati per mantenere i conflitti e seleziona variabili in modo casuale andando in seguito sulla base di una funzione ad accettare o rifiutare un nuovo assegnamento.

Questo algoritmo risulta essere un algoritmo stocastico di ricerca locale che basa la sua probabilità di accettare o meno una nuova assegnazione in base alla fase in cui ci troviamo, all’inizio (alte temperature) l’algoritmo sarà più propenso ad effettuare scelte più randomiche, mentre nella seconda fase (basse temperature) l’algoritmo sarà restio a cambiare spazio di ricerca locale e resterà nella stessa zona cercando di migliorare.

Ad ogni passo dell’algoritmo il passo di miglioramento sarà scelto valutando tramite una funzione che minimizza il numero dei conflitti se la nuova assegnazione ha un valore minore dell’attuale in tal caso sarà assegnata tale nuova assegnazione, se viceversa risulta avere un valore maggiore dato da un maggior numero di conflitti allora verrà eseguito un passo random ovvero tale assegnazione sarà accettata oppure rifiutata in base alla funzione data dalla distribuzione di **Gibbs-Botzmann**:

Logo, company name

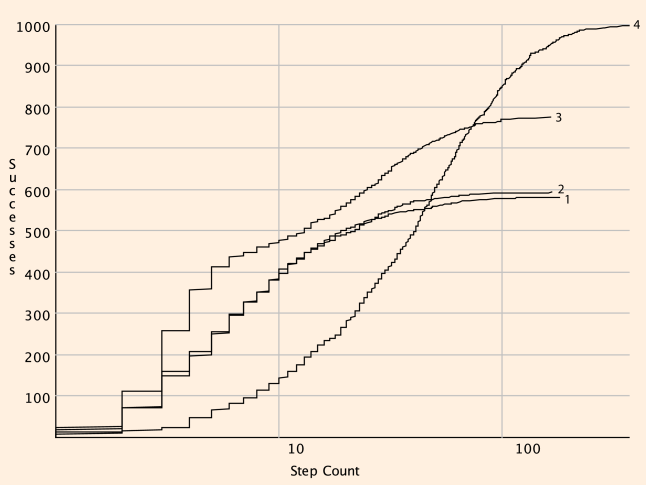
Description automatically generated

La probabilità è dettata dal annealing schedule ovvero uno schema di riduzione della temperatura che influenzerà la funzione di Gibbs-Boltzmann. Un approccio usato è il raffreddamento geometrico.

**VALUTAZIONE DEGLI ALGORITMI RANDOMIZZATI**

Il confronto tra algoritmi randomizzati non è semplice e risulta difficile specialmente quando non trovano una soluzione poiché in tale situazione bisogna decidere un opportuno criterio di stop senza che l’algoritmo cicli indefinitivamente.

Uno dei migliori modi per cercare di compararli è usare una **run-time distribution** ovvero plottare un grafico che mostri come l’algoritmo si comporta su ogni istanza data da risolvere.



**RANDOM RESTART**

Un algoritmo che ha una bassa percentuale di successo può essere migliorato rimandandolo in esecuzione più volte e aumentando il numero delle esecuzioni aumenta la sua probabilità di successo. Infatti, se un algoritmo randomizzato ha una probabilità di successo p e lo rieseguiamo n volte allora la sua probabilità di successo sarà data da :

1 – (1 - p)n

Per capire come sfruttare al meglio questa proprietà ci serve l’analisi condotta con il grafico del **run-time distribution**  visto in precedenza dal quale possiamo capire quali algoritmi risultino più adatti a tale processo di iterative random restart.

Altre volte alcuni algoritmi randomici possono essere più complessi e non del tutto adatti da iterare più volte a causa del costo computazionale in questi casi si preferisce avere un approccio a **partial restart**; tuttavia, qui non possiamo assumere l’indipendenza ogni qualvolta che facciamo un nuovo tentativo.

**POPULATION-BASED METHODS [algoritmi genetici basati sulle popolazioni]**

Negli algoritmi visti fino ad ora di ricerca locale dopo aver casualmente generato un assegnamento questo assegnamento viene portato avanti cercando di miglioralo, con gli algoritmi genetici basati sulle popolazioni si considerano più assegnazioni totali, le quali vengono portate avanti e quindi migliorate simultaneamente, un assegnazione totale in questi algoritmi prende il nome di individuo e un insieme di individui prende il nome di popolazione.

**BEAM SEARCH [ricerca per fascio]**

L’algoritmo è simile all’iterative best improvement, ma differisce per il fatto di sviluppare e portare avanti k assegnazioni totali per volta anziché una. Per cui ad ogni passo verranno portati avanti i k migliori successori generati delle precedenti assegnazioni. Se k fosse uguale ad 1 avremmo a che fare esattamente con un greedy descent.

**STOCHASTIC BEAM SEARCH**

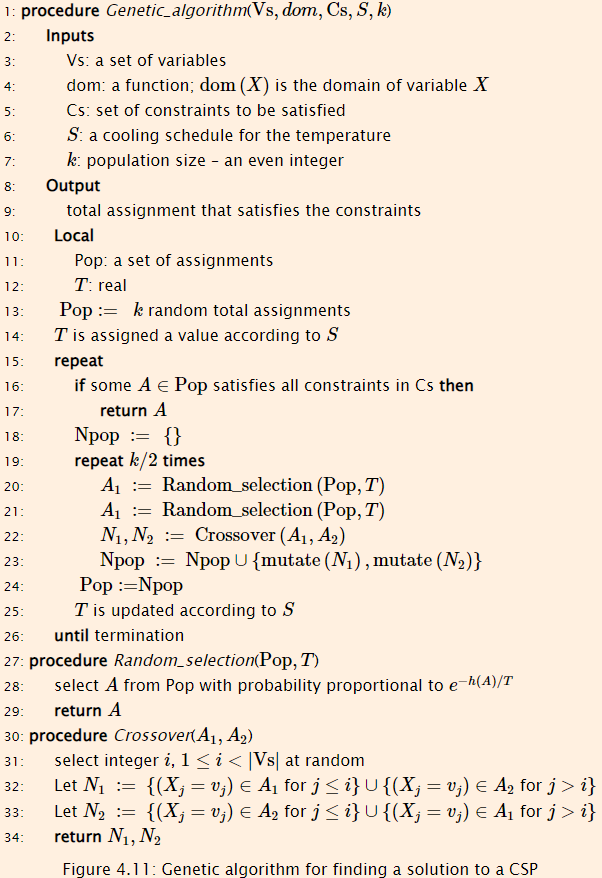
Invece di scegliere i migliori k successori secondo una funzione di valutazione i k successori sono scelti in modo randomico; tuttavia, **gli individui con una migliore valutazione hanno più probabilità di essere selezionati** dove il processo di assegnazione della probabilità con cui accettare una assegnazione è simile al simulated anneling ovvero segue la distribuzione di **Gibbs-Boltzmann**:

****

Dove h(A) è la funzione di valutazione e T è la temperatura. Questo approccio permette di avere una maggiore diversità tra le assegnazioni rispetto al beam search.

**GENETIC ALGORITHMS**

Sono simili agli algoritmi stochastic beam search, ma in questo caso ogni nuovo individuo deriva da una coppia degli individui della precedente popolazione e tali individui che generano i nuovi individui vengono detti parenti (o genitori). L’idea e di dividere un individuo genitore in due porzioni le quali saranno divise alla prole con un meccanismo chiamato **crossover**.



Line 20 e 21 A1 e A2 sono 2 assegnazioni casuali scelte con la funzione interna Random\_selection(). E saranno questi 2 individui a generare altri nuovi due individui, tale procedura è ripetuta k/2 volte ne consegue che per il funzionamento corretto k è sempre pari. Una volta scelti questi 2 individui la funzione Crossover() viene chiamate e eseguirà il crossover di A1 con A2 per generare i 2 figli della nuova generazione.

La procedura di crossover usata nell’algoritmo descritto prende il nome di **one-point crossover**  che consiste nel fissare un indice di split che dividerà i due genitori nello stesso punto e ricombinerà le parti divise unendole.

**OTTIMIZZAZIONE**

Invece di avere solo mondi possibili che soddisfano vincoli o meno, spesso abbiamo una relazione **di preferenza** sui mondi possibili e vogliamo un miglior mondo possibile in base alla preferenza, in tal caso siamo difronte ad un problema di ottimizzazione descritto:

* un insieme di variabili, ciascuna con un dominio associato
* una **funzione obiettivo** che mappa le assegnazioni totali a numeri reali
* un **criterio di ottimalità** , che consiste tipicamente nel trovare un'assegnazione totale che minimizzi o massimizzi la funzione obiettivo.

Un **problema di ottimizzazione vincolata** è un problema di ottimizzazione che ha anche vincoli rigidi che specificano quali assegnazioni alle variabili sono possibili. L'obiettivo è trovare un'assegnazione ottimale che soddisfi i vincoli rigidi.

Una tecnica di ottimizzazione è **la programmazione lineare** che risulta essere la classe di ottimizzazione vincolata in cui le variabili sono valori reali, la funzione obiettivo è una funzione lineare delle variabili e i vincoli rigidi sono disuguaglianze lineari.

I problemi di ottimizzazione hanno una difficoltà che va oltre i problemi di soddisfacimento dei vincoli. È difficile sapere se un compito è ottimale. Mentre, per un CSP, un algoritmo può verificare se un'assegnazione è una soluzione semplicemente considerando l'assegnazione e i vincoli, nei problemi di ottimizzazione un algoritmo può determinare se un'assegnazione è ottimale solo confrontandola con altre assegnazioni.