Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Didattica degli Algoritmi: studio di un esempio per la scuola primaria

Relatore:
Chiar.mo Prof.
RENZO DAVOLI

Presentata da:
FILIPPO BARTOLINI

Correlatore:
Dott.
MICHAEL LODI

Sessione III
Anno Accademico 2016-17

Alla mia famiglia e a tutti coloro
che hanno creduto in me

Introduzione

> *Le acquisizioni più valide nell'educazione scientifica e tecnologica sono quegli strumenti mentali di tipo generale che rimangono utili per tutta la vita. Ritengo che il linguaggio naturale e la matematica siano i due strumenti più importanti in questo senso, e l'informatica sia il terzo.*
>
> — George Forsythe

La pervasività dell'informatica e il suo essere destinata ad incidere sempre più significativamente in tutti gli ambiti della vita quotidiana, hanno reso necessario l'inserimento del suo insegnamento nei processi formativi. L'informatica va insegnata, studiata e compresa fin da bambini, perché la conoscenza dei suoi fondamenti contribuisce a formare la consapevolezza che le innovazioni a cui stiamo assistendo quotidianamente sono dovute al progresso di una disciplina autonoma, fondata su un insieme omogeneo e stabile di concetti, metodologie e competenze. Inoltre, essa ha un duplice ruolo nell'insegnamento: da una parte un ruolo *culturale e formativo* di disciplina scientifica di base; dall'altra un ruolo di *strumento concettuale trasversale* a tutte le discipline.

A questo proposito, un concetto ritenuto oggigiorno particolarmente significativo, nell'ambito della didattica dell'informatica, è il **pensiero computazionale**. Esso comprende un insieme di conoscenze, concettuali e metodologiche, e capacità di contestualizzazione applicativa che devono diventare patrimonio di tutti. Diversi sono i paesi europei che hanno integrato l'insegnamento della scienza informatica all'interno delle proprie politiche di educazione partendo dalla scuola primaria. Parte della didattica di questa

scienza viene svolta tramite il computer: strumento didattico proprio della disciplina.

In quest'ottica si colloca il presente lavoro di tesi. Partendo dall'esempio di un'attività per l'insegnamento dell'algoritmo del massimo, attraverso l'utilizzo di diversi ambienti di programmazione visuali a blocchi di tipo drag-and-drop, ispirati alla teoria costruzionista dell'apprendimento, si è ideato un modello progettuale *plugged* attuo all'insegnamento degli algoritmi a partire dal terzo anno della scuola primaria.

Essenzialmente ciò che questo lavoro si propone di fare è dare un'alternativa a quelle che sono le attività plugged attualmente disponibili (spesso uguali a quelle *unplugged*), mettendo però in risalto le caratteristiche chiave del *costruzionismo* e del *pensiero computazionale*.

Grande fonte di ispirazione per questa tesi è stata la pubblicazione *Imparare il pensiero computazionale, imparare a programmare* di Michael Lodi [51].

Struttura della tesi

Il primo capitolo presenterà una rassegna generale dello stato dell'arte nell'ambito della didattica dell'informatica, analizzando anche alcuni esempi a livello mondiale e italiano. Un primo aspetto originale è contenuto nella sezione 1.4.2 dove si mostrerà il confronto fra le *aree di conoscenza* coperte dai diversi strumenti *unplugged* rispetto al curriculum di informatica redatto da ACM/IEEE.

Nel secondo capitolo si analizzerà, inizialmente, il ruolo del gioco come strumento didattico. Successivamente verranno presentate le attività e gli ambienti di programmazione oggetto di questa tesi, con particolare attenzione alle caratteristiche positive e negative di questi ultimi.

Infine il terzo capitolo conterrà una serie di idee e consigli per la valutazione delle attività e gli sviluppi futuri.

Indice

Elenco delle figure

Capitolo 1

Stato dell'arte

> *I computer sono incredibilmente veloci, accurati e stupidi. Gli uomini sono incredibilmente lenti, inaccurati e intelligenti. L'insieme dei due costituisce una forza incalcolabile.*
>
> — A. Einstein

1.1 Era digitale

Durante tutto il periodo della *rivoluzione digitale* hanno iniziato ad esserci grandi cambiamenti nelle *Information and Communication Technologies* (ICT). L'uso della tecnologia ha assunto un'importanza sempre maggiore all'interno della *società dell'informazione*. La didattica nell'era digitale sta iniziando un lento cambiamento, passando da un modello di *pedagogia trasmissiva*, ovvero di tipo unilaterale, dove lo studente tende ad essere visto come un ricettore passivo e riproduttore di conoscenza, ad un modello di tipo *esperienziale e collaborativo*, dove si va ad incentivare l'apprendimento in situazione (*learning by doing*) facendo ricoprire allo studente la parte di protagonista attivo [58]. Anche la figura dell'insegnante, che trae le sue origini dalla *scuola della trasmissione*, inizia una lenta trasformazione: da *depositario assoluto del sapere* a *guida propositiva* tale da rendere il ricettore un costruttore attivo di conoscenza.

In questo contesto educativo hanno suscitato grande interesse le *teorie costruttiviste* che focalizzano l'attenzione sui processi cognitivi, sulle attitudini, sulla collaborazione attiva fra gli studenti e attribuiscono un ruolo meno centrale al docente, che viene visto piuttosto come facilitatore, ideatore e regista dei percorsi spontanei degli alunni [38]. Su queste teorie è basato il *costruzionismo* (*discovery learning)*, di cui Papert, matematico, informatico e pedagogista sudafriacano, è il padre.

Papert sostiene che l'apprendimento avviene in modo più efficiente se chi apprende è coinvolto nella produzione di oggetti tangibili, chiamati *artefatti cognitivi*. Tali artefatti concreti devono poter essere mostrati, discussi, esaminati, sondati e ammirati. Più in dettaglio, in base a questa teoria, la conoscenza non può essere trasmessa da un individuo all'altro, ma deve essere reinventata da ogni persona.

Il termine *costruzionismo* viene delineato nel libro *Constructionism: A new opportunity for elementary science education* [62], dove Papert lo definisce come una parola con due diversi aspetti: il primo che richiama le teorie costruttiviste di Piaget, che considera l'apprendimento come una ricostruzione e non come una mera trasmissione di conoscenze; il secondo afferma che l'apprendimento è più efficace e padroneggiato quando non è solo mentale, ma è supportato da una costruzione reale, come un progetto significativo. Quindi il costruzionismo è connesso all'apprendimento esperienziale, alla pedagogia dell'errore e ad alcune teorie di Piaget, ma con la peculiarità di dare grande rilevanza all'utilizzo degli artefatti.

1.1.1 Nativi Digitali

> Mia figlia maneggia telecomandi, tastiere e cellulari solo perché ne è circondata. Se la mia famiglia vivesse sull'isola di Chiloè, probabilmente sarebbe abilissima con le reti da pesca. È il contesto il motore delle abilità, non l'innata predisposizione. Non ci sono controprove del fatto che un australopiteco o un neanrdetaliano, dopo un po' di pratica, non sarebbero stati in grado di far

> scorrere le foto, con le dita, sul display di uno smartphone touch screen.
>
> (M. Albanese in [11])

L'espressione *nativo digitale* [65] identifica, secondo Prensky, una persona che è cresciuta con le tecnologie digitali, riferendosi in particolare a coloro nati negli USA dopo il 1985. Tutti coloro che sono nati prima di questa data vengono definiti dallo stesso scrittore come *immigrati digitali*. Tale data è stata scelta in quanto è l'anno che segna il passaggio cruciale verso la diffusione di massa dei computer.

Prensky afferma che i nativi digitali hanno sviluppato nuove connessioni neurali causate dall'uso delle ICT fin dall'infanzia e che questo li rende diversi nel modo di pensare e imparare [66]. In termini generali si è notata la presenza di una sorta di "media skill", ma in misura molto meno evidente rispetto a quanto acclamato da alcuni ricercatori per descrivere la *generazione digitale*. Si pensa che sia più corretto definirla nei termini di un'*attitudine*, ovvero una disposizione più marcata rispetto alle persone delle precedenti generazioni (*immigrati digitali*) nell'utilizzo delle tecnologie digitali.

Nel corso degli anni tante sono state le critiche nei confronti di Prensky e della sua espressione. Critiche che lo hanno portato a cambiare la locuzione in *saggezza digitale*, in modo da abbattere le barriere dell'età. Inoltre con questo termine non si va ad indicare che un nativo digitale abbia una maggiore abilità nel manipolare la tecnologia, ma solamente una maggiore capacità di prendere decisioni più sagge perché potenziate dall'uso della stessa.

Nonostante le critiche c'è ancora chi è a favore della tesi presentata da Prensky, sostenendo che siamo effettivamente di fronte ad una sorta di evoluzione mentale storica dell'umanità [36] [30].

1.2 Tecnologie didattiche e "formazione docenti"

All'interno di questo panorama si viene a delineare il settore di ricerca delle tecnologie didattiche che ha come obiettivo quello di studiare e analizzare il ruolo che esse hanno nell'innovazione dei processi di insegnamento e apprendimento, concentrandosi sull'educazione e non sulla tecnologia stessa [27]. La ricerca nel settore delle tecnologie didattiche ha carattere interdisciplinare poiché sviluppa ed elabora in modo autonomo contributi e modelli attinti da discipline diverse, come le scienze dell'informazione, la pedagogia, le scienze cognitive e la didattica multidisciplinare.

Negli ultimi anni l'introduzione delle tecnologie didattiche nelle scuole è aumentata sensibilmente, con l'obiettivo di favorire l'apprendimento da parte dei discenti. A questo proposito numerose ricerche hanno dimostrato che, da un punto di vista pedagogico, non ha alcun vantaggio introdurre dispositivi tecnologici nelle scuole se le strategie didattiche, le attività e il sistema scolastico non evolvono contemporaneamente [29]. Ancor più forti sono le critiche mosse dall'*Organizzazione per la Cooperazione e lo Sviluppo Economico* (OCSE) che, attraverso uno studio fatto nel 2015 [60], ha dimostrato che l'impatto dell'introduzione delle tecnologie didattiche sul rendimento scolastico non mostra alcuna evidente correlazione positiva. Per di più, nel medesimo studio, si mostra che l'utilizzo del digitale oltre un certo limite, diventa addirittura controproducente.

Facendo riferimento al caso specifico dell'insegnamento dell'informatica nelle scuole, alla luce di quanto sopra esposto, si può concludere che non è necessario introdurre in modo massiccio le tecnologie didattiche generali, come la *Lavagna Interattiva Multimediale* (LIM) o simili, se le lezioni rimangono di tipo frontale. Al contrario, il computer è una tecnologia didattica specifica della disciplina stessa e viene utilizzato per migliorarne l'apprendimento.

Sarebbe più opportuno concentrare le risorse a disposizione sulla formazione dei docenti nell'ambito della scienza informatica, invece che sull'utiliz-

zo delle tecnologie didattiche. Questo potrebbe realizzarsi tramite percorsi formativi obbligatori, istituiti sia sul campo che online, come ad esempio i *Massive Open Online Courses* (MOOC), classificati nella sezione *Online Education* in [44].

1.3 Pensiero Computazionale

> L'educazione ha due grandi uffizi: 1° di dar mano allo spiegarsi, al rinvigorire, al non deviare, all'operare regolato delle forza intrinseche della nostra spirituale natura; 2° di recare anticipatamente al fanciullo il soccorso di quelle cognizioni, di quelle norme e di quegli aiuti, che dalla società egli deve ricevere, per l'osservanza della legge morale del suo spirito in pro suo stesso, e in pro degli uomini suoi consorti.
>
> (R. Lambruschini in [47])

Uno degli scopi fondamentali dell'educazione è far comprendere i concetti fondamentali del sapere. La (scienza) informatica è così pervasiva al punto che farla comprendere è diventato fondamentale. Inoltre, inclusi nell'apprendimento dell'informatica, ci sono diversi valori socio-culturali che possono essere formativi indipendentemente dal fatto che vengano utilizzati immediatamente e in un campo piuttosto che in un altro.

A questo proposito, nel 2006, Jeannette M. Wing, scrisse un articolo [80] nel quale rese popolarmente nota l'espressione **pensiero computazionale** precedentemente introdotta nel 1980 da Seymour Papert nel libro *Mindstorms* [61]. Con il suddetto termine l'autrice intendeva sostenere che le capacità che vengono sviluppate dallo studio dell'informatica sono in realtà rilevanti per ogni attività umana. Brevemente, esso è la capacità logico-creativa di risolvere un qualsiasi problema pianificando una strategia:

> [...] pensare come un informatico, in modo algoritmico e a livelli multipli di astrazione.

Utilizzare il pensiero in modo computazionale significa suddividere il processo decisionale in singoli step e ragionare passo passo sul modo migliore per ottenere una soluzione per ognuno di essi, in modo tale che l'unione di queste soluzioni porti al raggiungimento dell'obiettivo prefissato.

Una delle caratteristiche principali del pensiero computazionale che la Wing descrive all'interno del suo articolo [80, p.35] è:

> [...] l'informatica non è programmare un computer. Pensare come uno scienziato informatico significa di più che essere in grado di programmare un computer. Esso richiede un pensiero a livelli multipli di astrazione.

Quindi l'obiettivo non è quello di formare una generazione di futuri programmatori, ma iniziare ad educare, fin da bambini, al pensiero computazionale.

D'altro canto, molti sostengono che sia difficile immaginare una maniera profonda di comprendere il pensiero computazionale se non studiando in qualche modo, a qualche livello, la programmazione, che permette di sperimentare come la comprensione della soluzione di un problema all'adatto livello di astrazione sia appunto un processo tipicamente umano.

Inoltre, nel corso degli anni, successivamente all'articolo introduttivo della Wing, si sono venute a creare diverse definizioni di *pensiero computazionale*. Alcune di queste, come osservato in [52], trattano il *pensiero computazionale* come un elemento nuovo e distaccato dall'informatica, risultando fuorvianti.

Alla luce di quanto detto sopra, una possibile definizione di *pensiero computazionale*, più generale e recente rispetto a quella della Wing, presente in [52], è:

> Il pensiero computazionale è l'insieme dei processi mentali usati per modellare una situazione e specificare i modi mediante i quali un agente elaboratore di informazioni può operare in modo effettivo all'interno della situazione stessa per raggiungere uno o più obiettivi forniti dall'esterno.

In conseguenza di ciò, se ne deduce che la materia che dovrebbe essere insegnata nelle scuole è effettivamente l'informatica.

Questo compito può risultare più arduo rispetto al passato dal momento che, da sempre, si è erroneamente abituati a considerare i computer come semplici macchine da utilizzare, limitando la conoscenza dell'informatica, per la maggior parte della popolazione, al loro mero utilizzo.

1.4 Strumenti

Gli strumenti didattici che si possono utilizzare per l'insegnamento dell'informatica, nel senso più ampio del termine, si possono suddividere in due macrocategorie:

1. *plugged*
2. *unplugged*

1.4.1 Plugged

Scratch e Code.org

Fra gli strumenti plugged più rilevanti troviamo Scratch [34], un ambiente di programmazione gratuito fondato su un linguaggio di programmazione a blocchi che permette di realizzare contenuti digitali interattivi (come giochi, storie e arte), ispirandosi alla *teoria costruzionista dell'apprendimento*. Tramite l'utilizzo di Scratch, consigliato a partire dagli 8 anni in su, si impara a pensare con creatività, a ragionare in modo sistematico, lavorare in maniera collaborativa e rende il successivo passaggio alla formalizzazione molto più immediato. Una delle sue peculiarità è il fatto che mette al centro la capacità del bambino di apprendere a esprimersi creativamente e a condividere le proprie creazioni con gli altri [70].

La creatività, che è considerata da Sir Kenneth Robinsons tanto importante quanto l'alfabetizzazione, è il processo che porta ad idee originali e di

valore. È un'abilità fondamentale che tutti quanti abbiamo fin da bambini, ma che il sistema educativo mondiale porta a disimparare [71] andando sempre di più verso la cultura della standardizzazione [72].

Oltre a Scratch, esistono anche delle organizzazioni no-profit volte a far insegnare a programmare (*learn to code*), come Code.org [3]. Esso è suddiviso sostanzialmente in due macro-sezioni: la prima dedicata agli insegnanti, nella quale, grazie a pagine a loro dedicate, si può richiedere l'aiuto dell'organizzazione stessa per portare la programmazione all'interno delle loro scuole; la seconda, dedicata agli studenti, nella quale sono disponibili 5 diversi corsi (4 corsi base più il corso rapido) per imparare la programmazione. Essi sono suddivisi in base all'età e all'esperienza pregressa dei discenti:

- Il **corso 1** è pre-scolare, progettato per essere utilizzato dai 4 anni in su, ma il cui uso è consigliato a partire dal primo anno della scuola primaria, in quanto è almeno necessaria l'iniziale capacità di lettura. I concetti fondamentali di programmazione che vengono coperti sono quelli di *sequenza* e *ripetizione* di istruzioni.

- Il **corso 2** è progettato per studenti dai 6 anni in su che hanno imparato a leggere, ma senza precedenti esperienze di programmazione. Il nuovo concetto di programmazione introdotto è quello di *istruzione condizionale*.

- Il **corso 3** è progettato per studenti dagli 8 anni in su che hanno svolto il corso 2. In esso si approfondiscono i concetti di programmazione precedentemente presentati e si introducono quelli di *funzione* e *ciclo "mentre"*.

- Il **corso 4**, nonché ultimo corso di quelli base, è progettato per essere utilizzato dagli studenti dai 10 anni in su che hanno svolto sia il corso 2 che il corso 3. Oltre al rafforzamento di quanto imparato fin'ora, si introducono i concetti di *variabili*, *cicli con contatori* e *funzioni con parametri*. In alternativa al corso 4 si può utilizzare in modo sostanzialmente equivalente il **corso rapido**, il quale fornisce una rapida

introduzione a tutti i concetti di programmazione presenti in tutti gli altri corsi.

La critica che viene fatta alle attività presenti in Code.org è il mancato risalto dell'espressione creativa. In diverse attività di introduzione al coding viene chiesto agli studenti di programmare i movimenti di un personaggio virtuale o di un piccolo robot [1] [8], che deve affrontare una serie di ostacoli per arrivare all'obiettivo finale. Questo approccio aiuta gli studenti ad apprendere diversi concetti base della programmazione, ma non permette loro di esprimersi in maniera creativa e/o di sviluppare un interesse duraturo per la programmazione [70]. Inoltre, per sviluppare la fluidità con la programmazione è necessario dare agli studenti l'opportunità di creare progetti e non solo risolvere degli enigmi logici.

Secondo le ricerche condotte dal MIT Media Lab l'introduzione alla programmazione dovrebbe seguire il principio delle **4 P** [69] [46]:

- **Projects**. Fornire l'opportunità ai bambini di lavorare su progetti significativi, cosicché riescano ad avere una visione completa del processo di trasformazione dall'idea iniziale fino alla creazione di qualcosa che possa poi essere condiviso con gli altri.

- **Peers**. Incoraggiare la collaborazione e la condivisone, in quanto la programmazione non dovrebbe essere un'atitvità solitaria.

- **Passion**. Consentire ai bambini di lavorare su progetti in linea con i loro interessi. Questo stimolerà il loro impegno e il loro apprendimento.

- **Play**. Incentivare i bambini a sperimentare attraverso il gioco (*playful experimentation).*

Princìpi ispirati dalle teorie di Papert e che sono ben strutturati e delineati all'interno di Scratch.

1.4.2 Unplugged

CS unplugged

CS unplugged [22] è, probabilmente, la raccolta di attività di apprendimento gratuito per l'insegnamento dei concetti chiave dell'informatica, attraverso giochi e puzzle, più utilizzata nel mondo, adatte per persone di ogni età. Esse spaziano dall'insegnamento dei numeri binari, alla crittografia, passando per lo studio di algoritmi, procedure e molto altro. Tutte le attività sono pensate per essere svolte in gruppi e si rifanno ai seguenti principi:

1. **No Computers Required**. Tutte le attività sono indipendenti dal computer. In questo modo si evita di confondere l'informatica con la programmazione e si rendono le attività disponibili a tutti coloro che non vogliono o non sono in grado di lavorare con il computer.

2. **Real Computer Science**. Le attività presentano i concetti fondamentali dell'informatica, sottolineando che la programmazione è un mezzo e non il fine.

3. **Learning by doing**. Le attività tendono ad essere cinestesiche, favorendo il lavoro di squadra e incoraggiando l'approccio *costruttivista*.

4. **Fun**. Le attività sono pensate e sviluppate per essere il più divertenti e coinvolgenti possibili. Spesso i problemi sono presentati come parte di una storia piuttosto che come una sfida matematica.

5. **No specialised equipment**. Le attrezzature necessarie per svolgere le attività sono a basso costo e spesso già presenti all'interno delle aule scolastiche.

6. **Variations encouraged**. Tutte le attività sono rilasciate sotto licenza *Creative Commons BY-NC-SA*, che consente la distribuzione, modifica e creazione di opere derivate dall'originale a condizione che venga riconosciuta una menzione di paternità adeguata e che alla nuova opera venga attribuita la stessa licenza dell'originale.

7. **For everyone**. Sono incoraggiate le variazioni alle attività per un miglior adattamento alla cultura locale, in quanto tutte sono pensate per essere *inclusive*.

8. **Co-operative**. Sono incentivate la cooperazione e la comunicazione per la risoluzione dei problemi.

9. **Stand-alone Activities**. Le attività sono divise in moduli che possono essere utilizzati indipendentemente l'uno dall'altro.

10. **Resilient**. Nonostante si commettano dei piccoli errori durante lo svolgimento delle attività, essi non impediscono ai partecipanti di comprenderne i principi chiave.

Inoltre, esse sono state sviluppate utilizzando il medesimo *design pattern* [59], così costituito:

- **Pattern Name**. Assegnazione un nome appropriato all'attività.
- **Problem**. Spiegazione del problema che deve essere risolto.
- **Context**. Il contesto o la situazione in cui la soluzione è applicabile, tenendo in particolare considerazione l'età dei bambini.
- **Forces**. Condizioni affinché questa attività sia applicabile: tipi di materiale, tempo a disposizione e conoscenze pregresse degli insegnanti.
- **Solution**. La strategia per risolvere il problema.
- **Resulting Context**. L'obiettivo è quello di incrementare il numero e la qualità delle attività disponibili cosicché gli educatori possano scegliere quella più appropriata per il concetto che intendono spiegare o illustrare.
- **Rationale**. Le risorse didattiche hanno lo scopo di insegnare l'essenza dei principi dell'informatica, anche se l'insegnante non ha familiarità con il concetto. I giochi e i lavori di gruppo migliorano la motivazione degli studenti impegnandoli a riflettere sui princípi dell'informatica.

Alcune sono le critiche mosse nei confronti delle attività di CS Unplugged. In particolare, una riguarda la seguente domanda: *Does the content of the activities reflect the content of CS?* Poiché queste attività mirano a formare gli studenti sull'informatica, un gruppo di ricercatori [76] ha verificato se esse rappresentino in modo corretto il tipo di pensiero coinvolto, utilizzando come metro di paragone il curriculum di informatica redatto da ACM/IEEE [31] che fornisce un quadro completo di ciò che gli esperti di formazione considerano di competenza dell'informatica. Delle 14 *Knowledge Areas* (KA) presenti nel suddetto curriculum, solo le 7 seguenti sono rappresentate nelle attività di CS Unplugged:

1. *Discrete Structures*
2. *Algorithms and Complexity*
3. *Architecture and Organization*
4. *Programming Languages*
5. *Net Centric Computing*
6. *Human-Computer Interaction*
7. *Intelligent System*

Tra le aree sopra elencate, due sono quelle particolarmente dense di attività: *Architecture and Organization* e *Algorithms and Complexity*, con, rispettivamente, 6 e 9 attività. Ci sono diverse altre aree facenti parte dell'informatica che non hanno attività che le rappresentano: alcune perché non sono adatte a questo tipo di metodo di sensibilizzazione, come per esempio *Software Engineering*, mentre altre potrebbe essere incluse, come ad esempio *Social and Professional Issues* e *Programming Fundamentals*.

Una seconda critica, presa in esame dal medesimo studio [76], è concerne alla seguente domanda: *Are the CS Unplugged activities explicitly linked to CS?* Ogni singola attività è divisa in tre macrosezioni: il sommario, la

spiegazione dell'attività con annessi i vari "fogli di lavoro" e la sezione "Cosa c'entra tutto questo?". Quest'ultima parte chiarisce la relazione che è presente fra l'attività stessa e l'informatica. Se si definisce il *livello di collegamento*, fra l'attività e l'informatica, come il numero di volte in cui esso è stato menzionato all'interno della stessa, si può vedere che solo 11 delle 24 attività totali hanno un collegamento esplicito, di cui 4 di livello alto, una di livello medio e le restanti di livello basso.

Entrambe le suddette critiche sono nate dall'analisi fatta ai seguenti 4 processi interconnessi, identificati in [48], che dovrebbero apparire nell'apprendimento di nuove conoscenze e nell'integrazione di esse con quelle preesistenti:

1. **Elicitare idee esistenti relative alla conoscenza acquisita**.
2. **Introduzione di nuova conoscenza**.
3. **Sviluppo di criteri per la valutazione della nuova conoscenza acquisita**.
4. **Ordinamento della conoscenza**.

Linn and Eylon in [48] sostengono che molti approcci didattici non riescono a supportare un'acquisizione di conoscenza stabile perché mancano in uno o più processi. Analizzando le attività di CS Unplugged si è venuto ad evidenziare che, per quanto riguarda il primo processo, solo 6 delle 24 attività totali includono una discussione iniziale, prima dell'attività stessa, che coinvolge la conoscenza preesistente degli studenti. La maggior parte delle attività si concentrano principalmente sul secondo processo, cioè quello di introduzione di nuove conoscenze, mentre gli ultimi due processi non si vengono quasi mai a sviluppare.

Anche Perkins in [63] concorda con quanto sostenuto da Linn and Eylon nel primo dei 4 suddetti processi. Durante lo svolgimento delle attività di CS Unplugged gli studenti acquisiscono nuove conoscenze in merito all'informatica che dovrebbero migliorare le loro idee, ma affinché ciò accada, è neces-

sario cercare di costruire delle relazioni tra le parti della nuova conoscenza allo scopo di renderla sia stabile che non rapidamente dimenticabile.

Nonostante le precedenti critiche, in un recente studio [42] si è dimostrato che l'apprendimento dell'informatica, nei bambini, è più efficiente se inizia attraverso lo svolgimento delle attività **unplugged** piuttosto che di quelle **plugged**. Per dimostrarlo si sono divisi i bambini di diverse classi, in scuole differenti, in due gruppi: il primo ha iniziato lo studio dei concetti di programmazione tramite attività unplugged, mentre il secondo attraverso quelle plugged.

I tre aspetti considerati nel confronto fra i due gruppi sono stati:

1. **facilitare la comprensione dei concetti di programmazione**;
2. **motivare e sostenere la "self-efficacy" degli studenti nei lavori di programmazione**;
3. **motivare gli studenti ad esplorare e usare costrutti di programmazione nei loro lavori**.

I concetti di programmazione che sono stati spiegati durante le 8 settimane di lezione sono:

- **loops**
- **conditionals**
- **procedures**
- **broadcasts**
- **parallelization**
- **variables**

Al termine delle lezioni si è notato che i bambini del primo gruppo hanno migliorato significativamente la propria "self-efficacy" e quando si è passati

all'utilizzo di strumenti plugged, in particolare Scratch, sono riusciti ad utilizzare un numero maggiore di blocchi diversi rispetto ai bambini del secondo gruppo.

Un ulteriore aspetto da considerare è il comportamento dei bambini, all'interno dell'aula, durante le lezioni. La presenza, fin da subito, dei computer in classe, per i bambini del secondo gruppo, è stato oggetto di forte distrazione e non ha contributo a migliorare lo svolgimento delle lezioni. Al contrario, questo effetto negativo non si è verificato sui bambini del primo gruppo.

Programma il Futuro

Il sito *Programma il Futuro* [9] presenta una serie di attività di coding che possono essere utilizzate a partire dal primo anno della scuola primaria. Esso è un'iniziativa del MIUR in collaborazione con il CINI e traduce attività tratte dal sito Code.org [3]. Fra i vari percorsi offerti da *Programma il Futuro* è presente una sezione denominata "Lezioni tradizionali", volta all'insegnamento dell'*informatica* attraverso attività unplugged orientate alla programmazione. Esse sono così suddivise:

- **Programmazione su carta a quadretti**, il cui obiettivo è far comprendere agli studenti cos'è effettivamente la programmazione.
- **Algoritmi**, il cui obiettivo è far capire l'importanza di rendere ogni istruzione di un programma chiara ed il più possibile non ambigua.
- **Funzioni**, il cui obiettivo è illustrare come azioni ripetitive all'interno di un programma possono essere scritte a parte e richiamate all'occorrenza nel programma principale.
- **Istruzioni condizionali**, il cui obiettivo è illustrare come l'evoluzione di un programma sia influenzata dal valore assunto da alcune variabili, utilizzate all'interno del programma stesso.
- **Composizione di canzoni**, richiama il concetto di funzione, ma in modo più completo rispetto alle lezioni precedenti.

- **Astrazione**, il cui obiettivo è portare a riflettere su come rimuovere i dettagli da qualcosa, in modo da renderlo il più generale possibile.
- **Programmazione a staffetta**, il cui obiettivo è spingere gli studenti a confrontarsi con il processo di scrittura di un programma e con quello di controllo della presenza di eventuali errori e della loro correzione (debugging).
- **Internet**, il cui obiettivo è spiegare alcune caratteristiche della tecnologia, la quale rende possibili internet e il *World Wide Web* (WWW).

Prendendo come metro di paragone il curriculum di informatica redatto da ACM/IEEE [31] e le suddette attività, si può notare che le KA coperte sono:

- *Programming Fundamentals*
- *Algorithms and Complexity*

Con particolare rilevanza quella di *Programming Fundamentals*.

Questo progetto è iniziato nell'anno scolastico 2014-15 ed è tutt'ora in corso. Leggendo nei diversi rapporti stilati alla fine di ogni anno scolastico e in vari articoli scientifici [33], si nota un incremento notevole nel numero di insegnati e di scuole che vi hanno partecipato, riportando una soddisfazione generale da parte degli insegnanti stessi.

Hello Ruby. Avventure nel mondo del coding.

Hello Ruby Adventures in Coding [50] è un libro scritto da Linda Liukas con l'obiettivo di introdurre i concetti fondamentali del pensiero computazionale ai bambini, in quanto, secondo l'autrice, in futuro tutti ne avranno bisogno a prescindere dal lavoro che andranno a svolgere. Il libro è stato progettato per essere letto insieme ai genitori ed è formato da due differenti parti. Nella prima parte viene descritta la storia di Ruby e di altri personaggi che lei incontra durante la sua avventura. Essi sono:

- **Tux il pinguino e i suoi amici**. Sono molto intelligenti, ma comunicano utilizzando frasi molto brevi. Amano i problemi, specialmente spezzandoli in pezzi più piccoli.
- **Leopardo delle nevi**. È il più bello ed educato, ma spesso litiga con i robot, nonostante la loro somiglianza.
- **Le volpi**. Amano essere entusiaste, amichevoli e allegre. Non si arrabbiano mai tranne quando si cerca di limitarne la libertà.
- **I robots**. Sono giocosi, flessibili e veloci. Hanno centinaia di fratelli e la cosa che li rende più felici è quando costruiscono qualcosa tutti insieme.
- **Django e il suo serpente domestico chiamato Python**. Sono molto organizzati e piuttosto rigidi. A loro piacciono molto le cose che possono essere contate.

Ognuno di questi personaggi rappresenta tecnologie esistenti in ambito informatico e la Liukas, in una breve descrizione, ne evidenzia alcuni pregi e alcuni difetti. Inoltre in ogni capitolo dell'avventura è contenuto un diverso concetto del pensiero computazionale.

Nella seconda parte sono presenti alcuni esercizi, sotto forma di giochi *unplugged*, attui sia ad insegnare i concetti del pensiero computazionale, inteso come propensione alla programmazione, sia a sviluppare la creatività dei bambini. Creatività che anche la Liukas, come Sir Kenneth Robinsons, ritiene una skill fondamentale che ogni persona al mondo dovrebbe apprendere. All'inizio di ogni attività è presente un *toolbox* in cui vengono evidenziati gli insegnamenti contenuti nei successivi esercizi. Svolgendo tutte le attività si impareranno i seguenti concetti di programmazione:

- **Sequence**
- **Decomposition**
- **Pattern Recognition**

- **Data type**
- **Algorithms**
- **Data structures**
- **Loop**
- **Selection**
- **Creativity and Technology**
- **Functions**
- **Abstractions**
- **Debugging**
- **Pair programming**

Facendo lo stesso tipo di confronto, come in 1.4.2, tra il curriculum di informatica redatto da ACM/IEEE [31] e le attività di *Hello Ruby*, si può osservare che le KA coperte sono:

- *Programming Fundamentals*
- *Algorithms and Complexity*

Con particolare rilevanza quella di *Programming Fundamentals*.

Differenze concettuali fra i diversi strumenti unplugged.

I tre diversi strumenti *unplugged* che sono stati analizzati hanno tutti il medesimo obiettivo: insegnare il *pensiero computazionale* ai bambini partendo dai primi anni della scuola primaria. Ciò per cui si differenziano è l'interpretazione che viene data a questo concetto e la conseguente modalità di raggiungimento dell'obiettivo. Infatti le attività presenti in *Hello Ruby* e quelle presenti in *Programma il Futuro* sono incentrate nell'insegnamento

dei princìpi della programmazione. Al contrario, le attività presenti in *CS Unplugged* si basano sui concetti di algoritmo, complessità, architettura ed organizzazione del computer.

1.5 Esempi di esperienze a livello mondiale

L'informatica per i gradi di educazione identificati come *K-12* cioè dalla scuola dell'infanzia alla fine della scuola secondaria di secondo grado, che negli USA ha durata quadriennale, ha generato, e continua a farlo, grande interesse. In particolare si presta molta attenzione all'insegnamento nell'intervallo K-8, nel quale i bambini, oltre alla conoscenza, formano anche opinioni e attitudini per il loro futuro ruolo nella società. Inoltre, l'insegnamento dell'informatica nella fascia K-8 è un buon motivo per incoraggiare sia lo sviluppo dei talenti creativi dei bambini sia l'apprendimento collaborativo. Purtroppo è bene ricordare che l'informatica non è ancora sviluppata come materia a sé stante dal momento che non è presente nelle indicazioni nazionali di tutti i Paesi, ma è in stretta connessione con tutte le altre discipline dato che ha anche un alto valore trasversale [52] [25].

Inoltre, all'interno della fascia K-8, i curriculum informatici sono spesso basati sul concetto di *IT fluency*, ovvero sulla capacità del singolo individuo, durante la sua intera vita, di capire ed utilizzare le nuove tecnologie man mano che queste evolvono e si diversificano da quelle originariamente apprese. Questo metodo generale di creare la conoscenza è in forte contrasto con la *IT literacy*, la quale rappresenta solamente l'abilità di usare una precisa e particolare tecnologia [39].

In tutto il mondo diverse sono le realtà che già da diversi anni hanno introdotto l'insegnamento dell'informatica come materia scolastica obbligatoria a partire dalla scuola primaria.

Una di queste è l'Inghilterra che, dall'anno scolastico 2014-15, ha reso l'insegnamento dell'informatica coattivo tramite una pubblicazione fatta dal Ministero dell'Istruzione nel 2013 [12].

In modo simile negli Stati Uniti, tramite l'atto legislativo *Every Students Succeeds Act* del 2015, è stata introdotta l'informatica tra le *materie per un'istruzione a tutto tondo.*

Inoltre nel panorama europeo diversi sono i Paesi che da alcuni anni hanno integrato o stanno cercando di integrare, nel proprio curriculum nazionale, il *pensiero computazionale* [13].

1.6 Esempi di esperienze a livello italiano

In Italia, la locuzione *Pensiero Computazionale*, dal 13 luglio 2015, è entrata a far parte di un testo normativo, ovvero nella legge 107, la cosiddetta legge della *Buona Scuola*, nell'articolo 1 comma 7 lettera h.

Inoltre, nel medesimo anno, il MIUR ha avviato due iniziative:

1. *Programma il Futuro*
2. *Animatori digitali*

La prima è stata spiegata in 1.4.2. La seconda è una nuova figura di riferimento, presente all'interno di ogni istituto scolastico, il cui compito consiste nell'attuare il *Piano Nazionale Scuola Digitale* (PNSD) nella propria scuola di appartenenza; è un docente di ruolo con spiccate capacità organizzative che dovrebbe organizzare attività e laboratori per formare la comunità scolastica e lavorare per la diffusione di una cultura digitale condivisa tra tutti i protagonisti del mondo dell'istruzione [57]. Prima di iniziare la sua attività, l'*animatore digitale* deve seguire un percorso di formazione organizzato dalle scuole capofila selezionate dagli Uffici scolastici regionali.

A questo punto si può sottolineare un importante aspetto che è tutt'ora oggetto di discussione, ovvero *usare metodologie didattiche arricchite dall'uso di strumenti tecnologici non è necessariamente sinonimo di innovazione.* Oltre a quanto argomentato in 1.2, l'adozione di nuovi ambienti di apprendimento può favorire gli studenti nell'avere un ruolo attivo in classe, ma, senza un quadro culturale di riferimento, che serve per orientare i giovani alla prese

con il digitale, rischia di diventare un esercizio vuoto e noioso. Chiaramente esistono molteplici metodologie per l'integrazione degli strumenti tecnologici in classe e ogni animatore digitale dovrà adottare quella più connaturale al contesto in cui si trova.

Ovviamente le ore di formazione previste dal PNSD sono poche per preparare adeguatamente questa nuova figura e tutti coloro che sono realmente interessati al processo di innovazione educativa devono dimostrasi attivi rispetto alle varie opportunità offerte online (MOOC).

1.7 Donne negli STEM

Gli ambiti scientifici e tecnologici quali *Science, Technology, Engineering and Mathematics* (STEM) sono quelli in cui maggiormente le donne soffrono il *gender divide*. Le cause principali di questo fenomeno sono da ricercare nelle resistenze culturali interne all'organizzazione, nella mancanza sul mercato di laureate nelle discipline tecnico-scientifiche e nello scarso interesse da parte delle donne verso le professioni legate a tematiche di IT.

Inoltre, secondo Reshma Saujani, il deficit di coraggio [74] che affligge molte giovani donne è la causa del perché sono sottorappresentate nelle STEM e in molti altri ambiti. Spesso si tende ad educare le donne alla perfezione, ma questo le rende terribilmente prudenti e le porta ad evitare qualsiasi rischio. Questo atteggiamento si ripercuote nella programmazione, in quanto essa richiede perseveranza, che a sua volta richiede imperfezione.

Secondo la media stimata dall'*Organizzazione per la cooperazione e lo sviluppo economico* (OCSE), in 35 Paesi europei meno di un laureato su 5 in informatica è donna [26].

Inoltre, una ricerca svolta da Microsoft su un campione di 11.500 donne, di età compresa fra gli 11 e i 30 anni, in 12 diversi Paesi europei, ha evidenziato che l'interesse nei confronti delle *STEM* è presente nelle donne verso gli 11 anni, ma inizia a diminuire radicalmente attorno ai 15. Una problematica che, come fa notare il professor Martin W. Bauer, collaboratore di questa ri-

cerca, riguarda la conformità alle aspettative sociali e gli stereotipi di genere, che continuano a indirizzare le scelte delle ragazze verso campi lontani dagli *STEM* [56].

Nonostante ciò, diverse sono le figure femminili di rilievo che, nel corso degli anni, hanno dato il loro contributo in diversi campi scientifici e tecnologici, compresa l'informatica. Fra di esse ricordiamo *Ada Byron Lovelace*, *Rosza Peter*, *Grace Murray Hopper* e tante altre. Per di più, nei primi corsi di laurea in informatica, istituiti in Italia, la percentuale delle studentesse era attorno al 25%, con picchi del 30%. Durante gli anni novanta questa partecipazione è iniziata a calare fino a raggiungere l'8% nel 2012 [64]. Percentuali simili sono state registrate in diverse univeristà europee.

> La programmazione non è solo "roba" da maschi. Basta impararla fin da bambine. [L. Liukas]

Di questo è convinta Linda Liukas, che reputa i codici dell'informatica [49] come l'alfabeto del Ventunesimo secolo e sottolinea che tutte le persone (donne, uomini, bambine e bambini) dovrebbero conoscere le basi della programmazione per poter continuare a comunicare.

Diverse sono le comunità non-profit che insegnano i fondamenti dei linguaggi di programmazione alle donne di tutto il mondo, fra di esse si rammentano:

1. **Rails Girls** [10], fondata nel 2010 da Linda Liukas.
2. **Django Girls** [4], attiva dal 2014 con più di 1000 volontari, hanno organizzato eventi in 82 diversi Paesi.
3. **Girls Who Code** [7], fondata nel 2012 da Reshma Saujani con gli obiettivi di sostenere e aumentare il numero di donne nell'informatica; ispirare, educare e fornire loro le capacitè informatiche per perseguire coraggiosamente le opportunità del Ventunesimo secolo.
4. **Black Girls Code** [2], organizzazione che dal 2011 si occupa di fornire un'educazione tecnologica per le ragazze afro-americane.

In Italia è presente **Girls Code It Better** [6], un progetto creato da *Man At Work* (MAW) per avvicinare le ragazze delle scuole secondarie di primo grado alla tecnologia. I corsi sono totalmente gratuiti per le scuole, che devono mettere a disposizione solamente la propria aula di informatica.

Inoltre dal 2013 è attivo un ulteriore progetto, chiamato **Nuvola Rosa**, di *Microsoft* e *GrowItUp*, in collaborazione con la *Fondazione Mondo Digitale* [5], per avvicinare le donne italiane alle carriere *STEM*.

Capitolo 2

Lavoro originale

Sembra che abbiamo raggiunto i limiti di ciò che è possibile ottenere con la tecnologia informatica, anche se bisogna stare attenti nel fare tali affermazioni, perché tendono a suonare abbastanza sciocche nell'arco di 5 anni.

— John Von Neumann (1949)

2.1 Perché questa tesi

L'obiettivo che ci si è posti è quello di mostrare un possibile approccio *plugged* (per una più completa trattazione si rimanda il lettore all'Appendice A) per l'insegnamento dell'algoritmo del massimo che permetta di rispondere ad alcune delle critiche mosse verso *CS Unplugged* e contemporaneamente di favorire l'apprendimento del *pensiero computazionale* tramite la programmazione. Le attività con le quali si cerca di raggiungere tale scopo sono rivolte in particolare agli studenti a partire dal terzo anno della scuola primaria fino al quinto. Tuttavia è possibile recuperare il codice delle attività che, essendo open source, le rende modificabili a seconda delle esigenze dell'insegnante e quindi adattabili anche per gli studenti dei primi anni della scuola secondaria di primo grado. Gli strumenti dei quali ci si è avvalsi per realizzare le suddette attività sono stati, in ordine cronologico, *Scratch*, *Snap!* [40], *BloP* [37] e *Scratch Microworlds* [35].

2.1.1 Risorse preesistenti

Fra le varie attività presenti nella sezione *Sorting Algorithms* dei CS Unplugged [22], una è quella che richiama implicitamente l'algoritmo del massimo: *ordinare i pesi*. Lo scopo di questa attività è di trovare il miglior metodo per ordinare un insieme di pesi, senza conoscerne a priori il peso, con il solo ausilio di una bilancia che ne permetta il confronto a due a due.

Anche questa attività, come altre, soffre delle critiche che sono state precedentemente analizzate nel capitolo 1.4.2. Nonostante ciò, per provare ad avere un miglior risultato nell'apprendimento dell'algoritmo del massimo, sarebbe consigliabile partire dalla suddetta attività unplugged prima di passare a quelle plugged.

2.2 Dimensione (video)ludica

Il gioco non è un passatempo, ma per i bambini è un lavoro vero e proprio, è la loro attività principale in quanto attraverso di essa imparano e quindi, imparando, crescono [28]. Secondo Johan Huizinga [43] la stessa cultura nasce dall'attività ludica: in ogni creazione dello spirito, in ogni astrazione c'è un un'attività di trasformazione e/o invenzione di metafore che non sono altro che giochi di/con parole, segni, forme, codici. Concetti che si possono, allo stesso modo, riconoscere all'interno dei più moderni videogiochi, unendo il concetto di **homo ludens**, di Huizinga, con quello di **homo videns**, della società industriale di massa.

Il principale scopo del gioco è il divertimento, la cui radice, divertere, porta ad accomunare il gioco alla *creatività* o *pensiero divergente*. Se ad esso ci aggiungiamo il fattore della multimedialità allora si arriva a quella che si potrebbe definire una **cooperazione educativa**: condivisione coniugata a creatività [24]. Inoltre i videogiochi soddisfano tre esigenze fondamentali:

1. la **necessità di autonomia**, in quanto i bambini sono portati a compiere delle scelte;

2. la **necessità di competenza**, per superare le sfide;

3. la **necessità di relazioni**, che aggiungono un valore sociale al gioco.

Potrebbe essere un grave errore sottrarre spazio e tempo al gioco in un mondo in cui si va sempre più verso l'omologazione e la globalizzazione della cultura.

Piaget, da una prospettiva epistemologica, vede il gioco come un elemento centrale nella formazione del simbolo nel bambino. Egli, inoltre, distingue tre principali modalità di gioco, corrispondenti a tre diversi stadi di sviluppo:

1. **Gioco di esercizio**, a partire dai primi mesi di vita fino ai 2 anni;

2. **Gioco simbolico**, dai 2 fino ai 7 anni;

3. **Gioco con regole**, dai 7 fino agli 11 anni.

Quelli che sono stati sviluppati e che verranno descritti nelle sezioni successive fanno parte della terza categoria, dove la comparsa delle regole determina la fine del gioco infantile, propriamente detto, e inaugura una fase di crescita, altamente educativa, in cui viene stimolato l'autocontrollo del bambino, la sua capacità di concentrazione, di memoria e molto altro.

2.3 Giochi

> [..] il gioco contiene tutte le tendenze evolutive in forma condensata ed è esso stesso una fonte principale di sviluppo.
>
> (L. S. Vygotskij in [79])

Si sono sviluppate due diverse tipologie di giochi: la prima è una **gara di macchinine** in cui si affrontano un bambino e il computer, mentre la seconda è una **sfida ai tiri liberi a basket** che si svolge fra due bambini, oppure fra l'insegnante e un bambino.

2.3.1 Prerequisiti

Sono state previste due differenti versioni di algoritmi per il calcolo del massimo all'interno del gioco **gara di macchinine**, con, rispettivamente, diversi livelli di difficoltà:

1. **calcolo del massimo fra tre variabili** [16]
2. **calcolo del massimo fra gli elementi di una lista** [14]

Nella prima versione (Figura 2.1), ovvero quella più semplice, si richiede che il discente abbia almeno un'idea di come funzionino i costrutti *if* e *if-then-else* e di come vengano gestite le variabili all'interno dei linguaggi di programmazione. Il medesimo algoritmo è presente anche nel gioco **sfida ai tiri liberi a basket**, dove in più si richiede una semplice gestione dell'input.

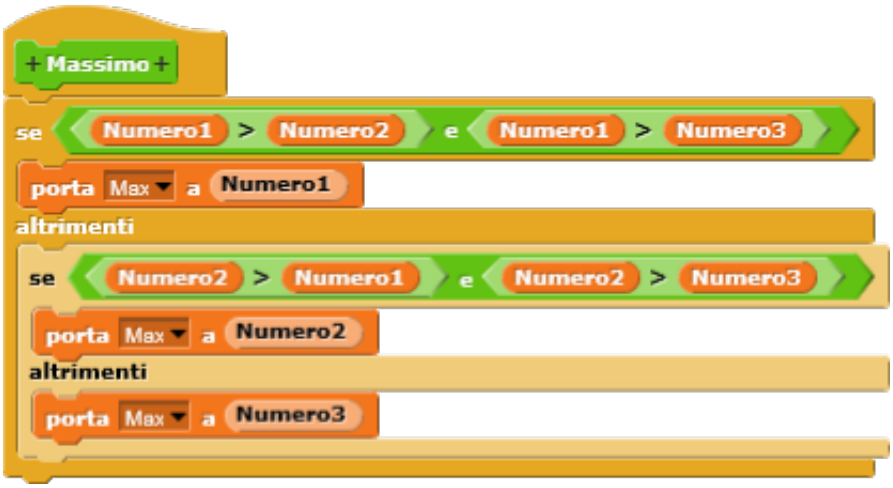

Figura 2.1: Algoritmo del massimo calcolato fra tre variabili

Nella seconda versione (Figura 2.2) è necessario ampliare la conoscenza acquisita dalla prima, integrandola con il costrutto di iterazione indeterminata *while*, il concetto di lista e la sua creazione e gestione.

Entrambe le attività sono state create utilizzando inizialmente Scratch, la cui spiegazione al suo iniziale utilizzo verrà fornita in 2.4.1, ed è quindi necessario aggiungere fra i vari prerequisiti una sua conoscenza di base: come si presenta l'interfaccia grafica e quali sono le aree che la compongono,

Figura 2.2: Algoritmo del massimo calcolato fra gli elementi di una lista

come funzionano i blocchi e in quali modi possono essere collegati fra di loro. Nello specifico conoscere e saper utilizzare buona parte delle *primitive* contenute nelle categorie **movimento**, **variabili e liste**, **situazioni**, **controllo**, **sensori** e **operatori**.

Inoltre, in entrambi i giochi sono presenti diversi *script*, oltre agli *sprite* (oggetti che eseguono determinate azioni all'interno di un progetto), *sfondi* e *costumi*. Tutti gli script potrebbero essere forniti, inizialmente, dall'insegnante agli studenti, in quanto richiedono un livello di conoscenza dei concetti di programmazione leggermente più elevato. Lo stesso discorso vale anche per quanto riguarda gli sprite, gli sfondi e i costumi perché così facendo si evita, almeno parzialmente, che gli alunni si perdano in questi dettagli grafici. L'unico script che non si deve fornire è quello per il calcolo del massimo.

Nello studio presente in [75] viene analizzato in quale momento e sotto quali condizioni una metodologia didattica sia da preferire rispetto ad altre. Questo è particolarmente rilevante per i modelli di apprendimento *costruzionisti* che enfatizzano la costruzione attiva della conoscenza da parte degli studenti. Uno degli aspetti presi in considerazione dal suddetto articolo è che gli studenti spesso non hanno avuto l'opportunità di sperimentare i tipi di problemi che sono resi risolvibili dalla conoscenza che viene loro insegnata.

Così facendo tratteranno la nuova conoscenza come fine a se stessa. Nei giochi che verranno successivamente illustrati si è reso l'algoritmo del massimo *necessario* per il loro funzionamento, cosicché gli studenti abbiano effettivamente un problema da risolvere, la cui soluzione sia l'algoritmo che vogliamo insegnare loro.

2.3.2 Illustrazione dei giochi

Gara di macchinine

I giochi [14] [16] si presentano con una interfaccia grafica intuitiva mostrata nella figura 2.3 e consistono in una gara fra due macchinine: una "comandata dal computer" e una "comandata dall'utente".

Figura 2.3: Interfaccia di gioco - gara di macchinine

L'obiettivo del gioco è tagliare il traguardo per primi. Per far avanzare la propria macchinina si deve, nella prima versione, trovare il valore massimo

fra tre diverse variabili; nella seconda trovare il valore massimo in una lista di interi. In entrambe prima che lo faccia il computer, altrimenti si perderà la gara. All'interno della funzione per il calcolo del massimo eseguita dalla macchinina comandata dal computer, è presente un'istruzione che la sospende per alcuni secondi prima di esplicitare il risultato. Questo ha lo scopo di permettere all'utente di avere un certo quantitativo di tempo per riuscire a rispondere correttamente prima che lo faccia il computer. Riducendo il tempo di sospensione della funzione, si renderà il gioco sempre più difficile. Tramite questo meccanismo si potrebbero anche fare delle semplici considerazioni sull'*efficienza* dell'algoritmo.

Sfida ai tiri liberi a Basket

Il gioco [19] mostra una semplice interfaccia grafica, come si vede dalla figura 2.4.

Figura 2.4: Interfaccia di gioco - sfida ai tiri liberi a basket

Lo scopo del gioco per il primo giocatore è quello di lanciare la palla per fare canestro tramite l'inserimento di tre numeri in input, il più difficili possibili, in modo che l'avversario non riesca a individuarne celermente il massimo. Per il secondo giocatore l'obiettivo è di individuare e inserire velocemente il massimo così da cercare di stoppare la palla ed evitare il canestro.

2.4 Versioni dei giochi

2.4.1 Scratch e Snap!

Scratch fornisce, oltre a quanto detto in 1.4.1, un metodo di programmazione visuale di tipo *drag-and-drop*: gli utenti "trascinano" i vari blocchi dalla *tavolozza dei blocchi* fino *all'area degli script*, dove possono collegarli fra di loro, come in un puzzle. Le strutture formate da più blocchi costituiscono uno **script**. Esistono tre diversi tipi di blocchi:

1. I **Blocchi impilabili**, la cui forma presenta un punto sul fondo che fuoriesce e/o uno in cima che rientra. Questi blocchi possono essere impilati fra di loro per formare delle sequenze e sono gli unici che, in alcuni casi, presentano un'area di input. Hanno colori diversi in base alla categoria di primitive d'appartenenza.

Figura 2.5: Blocco impilabile

2. I **Cappelli**, la cui parte superiore è arrotondata consentendogli di essere utilizzati solamente all'inizio di uno script.

Figura 2.6: Blocco a cappello

3. I **Reporter** che hanno una forma particolare che permette loro di essere inseriti all'interno delle aree di input presenti in altri blocchi.

Figura 2.7: Blocco reporter - primo tipo

Figura 2.8: Blocco reporter - secondo tipo

Gli ambienti di programmazione visuale, come Scratch e Snap!, vengono ampiamente utilizzati per introdurre i giovani discenti all'informatica e alla programmazione, incoraggiando l'apprendimento per scoperta (*learning by exploration*). Alcuni lati negativi di questo tipo di approccio vengono evidenziati in [55], dove gli autori sostengono che tale criterio generi abitudini di programmazione sbagliate. In particolare, vanno ad analizzare due aspetti della programmazione:

1. **metodologie di sviluppo software**
2. **granularità**

La critica riguardo al primo aspetto è che l'ambiente di programmazione di tipo visuale favorisca metodologie di sviluppo software totalmente *bottom-up*, partendo dai singoli componenti che vengono poi collegati insieme per formare un sistema sempre più compatto. Questo approccio, in linea con la filosofia del costruttivismo di Papert, se usato correttamente, consente ad un programmatore di progettare, implementare e testare componenti logicamente coerenti che possono essere integrate per sviluppare un sistema sempre più complesso. Al contrario, un suo uso sbagliato porta a non pensare più in modo algoritmico e a livello di progettazione del software, ma semplicemente a "trascinare" i blocchi che sembrano più appropriati per risolvere l'attività all'interno *dell'area degli script*, cercando di trovarne una possibile

combinazione. Secondo gli autori, Scratch, favorisce l'uso sbagliato di questa metodologia in quanto tutte le *primitive* sono presenti all'interno della *tavolozza dei blocchi* e quindi gli utenti non devono nè ricordarle a memoria nè decidere quali fra esse siano quelle necessarie. Per di più c'è la possibilità di lasciare frammenti di script o singole istruzioni all'interno *dell'area di script* senza che essi influenzino l'esecuzione del programma. La conseguenza di un'utilizzo sbagliato dell'approccio *bottom-up*, ovvero la critica al secondo aspetto, è l'incoraggiamento ad una programmazione "a grana estremamente fine" [55] e una programmazione "per bricolage" [78]. Scratch permette agevolmente la decomposizione degli script all'interno dei vari sprite, così da permettere la creazione di codice *modulare*, principio fondamentale di una buona progettazione software. Tuttavia un uso massiccio della decomposizione porta alla creazione di unità talmente ridotte da mancare, il più delle volte, di qualsiasi coerenza logica. Due sono le conseguenze che ne derivano. La prima è quella che permette di creare programmi qualitativamente mal strutturati (*spaghetti code)*, incomprensibili e, spesso, con un numero molto elevato di *script*. La seconda riguarda la gestione della concorrenza, fortemente incoraggiata in Scratch. La sincronizzazione dei vari script avviene tramite *message passing* che, unita ad una programmazione a grana estremamente fine, porta ad avere una difficile comprensione dell'esecuzione del programma, la possibile presenza di *race condition* e complica qualsiasi azione di *debug*.

Contrariamente a quanto sopra, in [54] viene dimostrato come il sistema di controllo della concorrenza di Scratch permetta di evitare la maggior parte delle *race condition*, nonostante non siano presenti meccanismi espliciti per la sua gestione come *semafori*, *locks* o *monitor*. Nel modello di Scratch, un *thread switch* è vincolato per poter essere fatto solo in due condizioni:

1. **su un comando che attende esplicitamente**, ad esempio *Wait 1 second*

2. **alla fine di un ciclo**

Esso non potrà mai avvenire nel mezzo di una sequenza di stati di non attesa o, per esempio, fra la condizione di un if e il suo corpo. Questo meccanismo permette agli utenti di ragionare su ogni script in modo isolato, prestando minor attenzione ai suoi *side effects*. L'unico e più comune caso di *race condition* che si possa verificare è quando più script vengono attivati da un evento *broadcast* e l'ordine di esecuzione di tali script non è quello che l'utente si aspetta.

Inoltre lo stile di *programmazione parallela* presente in Scratch, va in contrasto con la *programmazione sequenziale* che spesso viene preferita all'interno della didattica dell'informatica sia a livello universitario che a livello K-12. Ciò non ostante, in un recente articolo [45] viene sostenuta la tesi per la quale sia meglio insegnare ai nuovi studenti a pensare a programmare in parallelo in quanto non solo li preparerà meglio a programmare, in futuro, i nuovi dispositivi con *processori multicore*, ma aiuta anche a formare le loro menti a pensare in modo astratto piuttosto che semplicemente in termini di riscrittura del codice.

Tuttavia H. Lynn, fondatore di *theCoderSchool* [53], nel medesimo articolo, asserisce che sia prima necessario partire dalla programmazione sequenziale per permettere l'affermarsi delle abilità del *pensiero logico sequenziale*.

Durante lo sviluppo dei giochi, precedentemente illustrati in 2.3.2, tramite Scratch, ci si è soffermati sulla mancanza di alcune caratteristiche chiave, dal punto di vista didattico, di questo ambiente di programmazione. In particolare si è risentito della mancata possibilità di nascondere le definizioni di un *nuovo blocco* dalla *stage area* e degli *sprite* e gli *sfondi*, dalla zona sottostante, per far sì che il bambino si concentri, almeno inizialmente, solo sulla scrittura dell'algoritmo del massimo e non si distragga con il resto del codice o con i vari sprite. Per di più non è nemmeno possibile nascondere le *primitive* che Scratch mette a disposizione all'interno delle varie categorie. Inoltre non è contemplata la possibilità di "bloccare" la *Graphical User Interface* (GUI) così da evitare lo spostamento degli sprite e dello sfondo all'interno della *stage area* e la conseguente perdita di concentrazione da parte

dei discenti.

In seguito a queste considerazioni, si è passati a sviluppare i giochi [15] [17] [20] utilizzando *Snap!* per sfruttare alcune delle sue peculiarità.

Caratteristiche di Snap!

In questa sezione si analizzeranno alcune delle peculiarità di *Snap!* [41] che sono state utilizzate durante lo sviluppo dei giochi e che hanno un ruolo importante all'interno di questa tesi.

Snap! è basato su Scratch dal quale eredita tutte le principali caratteristiche, sia positive che negative, di *ambiente di programmazione visuale* di tipo *drag-and-drop*. Queste vengono integrate con alcune sue peculiarità che permettono di espandere le funzionalità di Scratch, così da poter essere utilizzato, non solo da studenti principianti, ma anche da quelli più avanzati, permettendo l'insegnamento di concetti di programmazione più complessi. La GUI di *Snap!* è molto simile a quella di Scratch, in quanto anch'essa è suddivisa in cinque differenti sezioni:

- **Toolbar**
- **Palette**
- **Scripting Area**
- **Stage Area**
- **Sprite Corral**

Nella *toolbar* sono presenti due bottoni in più rispetto a Scratch: *pause button* e *visible stepping button*. Il primo permette di sospendere manualmente uno script ed è particolarmente utile in quanto consente, durante la pausa, di eseguire altri script, controllare i valori delle variabili e modificare tutti gli script presenti nel programma, compreso quello in pausa. Il secondo bottone mostra l'esecuzione degli script istruzione per istruzione, così da rendere più semplice ed immediata l'azione di *debug*.

Il numero di primitive presenti in *Snap!* (versione 4.1) è molto maggiore rispetto a quelle di *Scratch* e tutte possono essere nascoste dalla *palette*. Tutti i *tipi di dato* presenti in *Snap!* sono *first class*, in quanto essi possono essere:

- *un valore di una variabile*
- *un input per una procedura*
- *un valore restituito da una procedura*
- *un componente di un aggregato di dati*
- *anonimi*

La peculiarità sulla quale ci si è concentrati maggiormente, durante lo sviluppo dei giochi, è stata la possibilità di costruire nuovi blocchi. *Snap!* permette, durante la costruzione di un nuovo blocco, di scegliere la categoria nella quale andare a inserirlo fra le dieci disponibili; il tipo di forma che il blocco assumerà e il nome. Sono presenti tre diversi tipi di forme che posso assegnare ad un nuovo blocco:

1. **Jigsawpuzzle-piece**, per un blocco di tipo comando;
2. **Oval block**, per un blocco di tipo monitor;
3. **Hexagonal block**, per un blocco di tipo condizione.

Successivamente alla scelta della forma del nuovo blocco, si ha la possibilità, tramite il *block editor*, di personalizzarlo. Il codice che comporrà il nuovo blocco non verrà visualizzato in nessuna area della GUI; l'unica cosa visibile sarà il nuovo blocco con la forma precedentemente selezionata e il suo nome, all'interno della *palette*.

L'unione di tutte queste caratteristiche ha permesso di colmare alcune delle lacune che sono presenti in Scratch, così da migliorare, dal punto di vista didattico, i due giochi per l'insegnamento dell'*algoritmo del massimo* e, più in generale, qualsiasi tipo di attività. In particolare, la presenza dei

due bottoni, *pause button* e *visible stepping button*, rende possibile all'insegnante mostrare agli alunni come avviene effettivamente l'esecuzione di un algoritmo passo passo, come individuare e successivamente correggere i possibili errori di programmazione. Inoltre anche gli alunni potranno sfruttare le caratteristiche di questi bottoni quando, in un secondo momento, modificheranno i giochi, personalizzandoli. La creazione di nuovi blocchi permette di "nascondere" alcune parti di codice in modo che i discenti si concentrino, in un primo momento, esclusivamente sulla scrittura del codice del massimo nelle differenti versioni. Per di più, grazie anche alla possibilità di nascondere le primitive delle varie categorie, l'insegnante può lasciare visibili solamente quelle che riterrà necessarie per scrivere il suddetto algoritmo.

Nonostante *Snap!* presenti una quantità di funzionalità che lo rendono *Turing-completo* e molto versatile, dal punto di vista didattico, potrebbe essere ulteriormente migliorato, soprattutto per quanto riguarda la GUI. Nella fattispecie, come anche in *Scratch*, è sempre possibile effettuare qualsiasi interazione con gli *sprite* e con gli *sfondi*, sia nella *Sprite Corral* che nella *Stage Area*, anche durante l'esecuzione di un programma. Sarebbe invece interessante avere la possibilità di "bloccare" la GUI in modo che i discenti non si concentrino, in un primo momento, sugli aspetti grafici, allontanandosi da quello che è l'effettivo scopo dell'attività.

Inoltre è bene ricordare che, nonostante sia possibile creare nuovi blocchi per non rendere visibili alcune parti di codice e nascondere le primitive, questo non implica che il discente non possa comunque trovare il modo per renderle nuovamente visibili, dal momento che è possibile qualsiasi tipo di interazione con l'interfaccia di *Snap!*.

Successivamente a queste osservazioni si sono sviluppati i giochi utilizzando un ulteriore ambiente di sviluppo: *BloP*.

2.4.2 BloP

BloP è un ambiente di sviluppo visuale basato su *Snap!* (versione 07/2013), dal quale eredita tutte le caratteristiche di base. La caratteristica principale

per la quale è stato sviluppato è di permettere agli insegnanti di creare delle versioni di qualsiasi linguaggio di programmazione tramite uno stile *a blocchi*, sostanzialmente uguale a quello presente sia in *Scratch* che in *Snap!*.

Ciò che lo rende particolarmente interessante ai fini di questa tesi è la possibilità di passare da Snap! a BloP e viceversa, riuscendo così a "sbloccare" l'interfaccia di Snap!, decidere che cosa mostrare e che cosa nascondere e "ribloccare" l'interfaccia. Nello specifico, quando la GUI è sbloccata si ha a disposizione Snap! con tutte le sue funzionalità, mentre quando è bloccata si passa a *BloP*, ottenendo un ambiente di sviluppo con le seguenti caratteristiche:

- tutti gli elementi quali *sprite*, *variabili* e *liste* presenti all'interno della *Stage Area* non sono nè trascinabili e nè modificabili;
- premendo sulla bandierina verde si inizia l'esecuzione dello script posizionato più in alto nella *Scripting Area* dello *sprite* selezionato;
- non è permesso avere *sprite* con lo stesso nome;
- non è possibile vedere alcun tipo di informazione sugli *sprite*, come lo stile di rotazione o la possibilità di essere trascinati;
- non è possibile nascondere o mostrare nè le primitive delle varie categorie nè le categorie stesse;
- non è possibile definire o rimuovere i blocchi personalizzati;
- non è possibile nè modificare le impostazioni di *Snap!* nè utilizzare il salvataggio e la condivisione sul cloud;
- le schede dei costumi e dei suoni di ogni *sprite* sono nascoste;
- non è possibile aggiungere ulteriori *sprite*.

Grazie alla peculiarità di *BloP* di riuscire a bloccare la GUI si riescono a risolvere la maggior parte delle mancanze legate a questo aspetto presenti sia in *Scratch* che in *Snap!*. In generale durante lo svolgimento di una

qualsiasi attività legata all'insegnamento di particolari concetti di programmazione, avere la possibilità di impedire agli alunni, in un primo momento, di apportarne modifiche grafiche, gli permette di concentrarsi esclusivamente su quelli che sono gli insegnamenti alla base della stessa. Nel caso specifico di questa tesi coniugando la creazione di nuovi blocchi attui a "nascondere" parte del codice, tramite *Snap!*, e la possibilità di bloccare la GUI tramite *BloP*, si inducono gli studenti a porre una maggior attenzione sulla scrittura dell'algoritmo del massimo, evitando così la maggior parte delle forme di distrazione.

Inoltre *BloP* presenta tre nuovi blocchi, disponibili anche quando si passa a *Snap!*, all'interno della categoria *Controllo*:

1. **Durante il caricamento**. Ha la forma "a cappello" ed esegue lo script posto sotto di lui durante il caricamento di un programma in *Snap!* o in *BloP*;

2. **Quando si esegue uno script**. Ha la forma "a cappello" e il codice scritto sotto di lui viene eseguito in *BloP* prima dell'esecuzione di ogni altro script (cioè quando si preme sulla bandierina verde o su un altro script presente nel programma);

3. **Esegui altri script**. È un "blocco impilabile" che viene utilizzato alla fine di uno script che inizia con il blocco "a cappello" *quando si esegue uno script*.

Il secondo ed il terzo blocco sopra elencati sono necessari ogniqualvolta si cerca di eseguire un programma in *BloP*, nel quale non siano stati precedentemente caricati dei blocchi di uno specifico linguaggio di programmazione.

Di contro però, da un punto di vista didattico, alcune delle caratteristiche introdotte in *BloP* dovrebbero essere riviste. Per una più completa trattazione si rimanda al capitolo 3.2.

2.4.3 Scratch Microworlds

I "micromondi" sono una recente versione personalizzata dell'editor di Scratch, realizzati per rendere più facile l'avvicinamento alla programmazione da parte dei principianti. Essi sono una combinazione di due differenti approcci adottati per l'apprendimento: quello basato sui puzzle (puzzle-based) e quello basato sui progetti (project-based) [77].

	Puzzle-Based	Project-Based		Scratch Microworlds
Focus	solve puzzle	**make project**		**make project**
Outcome	one right answer	**no wrong answers**		**no wrong answers**
Potential for Exploration	few possibilities	**many possibilities**		**many possibilities**
Potential for Creativity	single solution	**diverse projects supported**		**diverse projects supported**
Number of Coding Blocks	**a few blocks**	many blocks		**a few blocks**
Interface Design	**simple**	complex		**simple**
Scaffolding	**scaffolded**	not scaffolded		**scaffolded**
Learning Pathway	designer-led	**learner-led**		**learner-led**

Figura 2.9: Caratteristiche dei micromondi di Scratch [77]

I "micromondi" mantengono tutte le principali caratteristiche progettuali di Scratch, unite ad una semplice interfaccia e ad un limitato sottoinsieme di blocchi focalizzati su un interesse specifico (interest-based), così che gli studenti possano scegliere di lavorare su un argomento personalmente significativo per loro. A questo proposito si potrebbero considerare gli interessi dei due giochi per calcolare il massimo, rispettivamente, come due sport: una *gara di macchine* e *una partita di basket*. Il progetto Scratch Microworlds

fa parte dell'iniziativa "Coding for All: Interest-Driven Trajectories to Computational Fluency" e mira ad espandere la portata di Scratch a tutti quei gruppi che sono sottorappresentati nell'informatica, basandosi sugli interessi e le identità sociali dei bambini [67].

Ciò che si è provato a fare con i due giochi per l'insegnamento dell'algoritmo del massimo [21] [18] è stato di eliminare l'algoritmo stesso e di mostrare solamente un ridotto insieme di blocchi che ne permettano la ricostruzione. Così facendo i bambini sono obbligati a pensare, sfruttando i blocchi a loro disposizione, a come ricostruire l'algoritmo del massimo, solamente nella versione con tre variabili, cosicché il gioco ritorni a funzionare.

Chiaramente essendo l'editor di Scratch microworlds una modifica di quello di Scratch, risente anch'esso di tutti i suoi pregi e difetti precedentemente analizzati in 1.4.1 e 2.4.1.

2.5 Significato del gioco

Il concetto di *artefatti cognitivi*, precedentemente analizzato in 1.1, trova la sua più ovvia applicazione nel campo dell'insegnamento. In particolare, nell'*e-learning*, ovvero in quella branca dell'insegnamento che fa uso di tecnologie multimediali e di internet per migliorare la qualità dell'apprendimento. Entrambi i giochi presenti in 2.3 sono considerabili artefatti cognitivi, in quanto ne soddisfano tutte le caratteristiche. I giochi racchiudono al loro interno diversi significati educativi: nella costruzione, nella partecipazione, nella discussione e analisi, nell'esposizione e nello smontaggio e ricostruzione. La costruzione stimola fortemente la creatività degli studenti, il loro sviluppo cognitivo e migliora la capacità di pensare in modo sistematico; la partecipazione li sprona a trovare soluzioni, più o meno efficienti, per riuscire a vincere tramite l'elaborazione di una strategia o semplicemente procedendo per *prove ed errori* facendo si che l'apprendimento avvenga in modo graduale. La discussione, l'analisi e l'esposizione soddisfano tre importanti requisiti del pensiero computazionale: la formulazione del problema (*astrazione*),

l'espressione della soluzione (*automazione*) e l'esecuzione della soluzione e valutazione della stessa (*analisi*).

2.6 Valutazione del gioco

Una volta che i discenti si sono cimentati nella scrittura dell'algoritmo del massimo all'interno di uno dei due giochi, in una delle versioni proposte, spetta all'insegnante valutarne l'effettiva comprensione. Analizziamo di seguito tre diverse metodologie di valutazione: due "plugged" e una "unplugged".

2.6.1 Plugged

Il primo criterio di valutazione plugged consiste nel far scrivere agli studenti, che sono riusciti a costruire in modo corretto l'algoritmo del massimo, quello del *minimo* per il medesimo gioco. In questo modo si andrà ad osservare se gli studenti hanno realmente compreso come costruire, in generale, un algoritmo e come calcolare il massimo e il minimo. Inoltre si rafforzeranno i costrutti di programmazione quali *if* e *if-then-else* e migliorerà quella che è la gestione delle variabili all'interno dei linguaggi di programmazione.

Il secondo criterio, che può essere anche conseguente al primo, è di modificare o ricreare altre parti del gioco, a discrezione dell'insegnante in base al livello acquisito dai discenti, cosicché possano personalizzarlo. Questo è possibile in quanto tutti i giochi presentano codice *open source* facilmente recuperabile.

In entrambi i casi, se lo studente non fosse riuscito a scrivere l'algoritmo del massimo nel modo corretto, dovrebbe chiedere un aiuto o un consiglio prima altri compagni e successivamente all'insegnante, la quale dovrebbe indirizzarlo sulla strada corretta senza però dargli direttamente la soluzione "giusta".

2.6.2 Unplugged

Utilizzando, al contrario, un metodo di valutazione unplugged, si interrogheranno gli studenti sul procedimento, sul metodo di sviluppo e sul ragionamento che hanno utilizzato per calcolare il massimo e costruirne l'algoritmo.

Un esempio su come si possa trovare il valore massimo fra due differenti numeri è quello di contare il loro numero di cifre, decretando come massimo quello che ne ha un numero maggiore. Nel caso in cui abbiano lo stesso numero di cifre, si andranno a confrontare a due a due partendo da quelle più significative.

Capitolo 3

Valutazioni e sviluppi futuri

La vita è un insieme di costrutti "sequenza" e "if... then... else". Qualcuno si è dimenticato il più importante, il "while... do", per tornare indietro.

— A. Balestrero
Applicazione del Teorema di Bohm-Jacopini

3.1 Valutazioni

Per raggiungere l'obiettivo prefissato in 2.1 si sono create due *attività plugged* utilizzando un approccio sostanzialmente differente rispetto a quelli che sono stati, e che continuano ad essere utilizzati per l'insegnamento di algoritmi specifici e, più in generale, per la programmazione.

Ciò che viene fatto nei suddetti approcci è di ricreare le attività *unplugged* tramite l'utilizzo di strumenti *plugged* oppure di predisporre degli esercizi di programmazione o di logica, fini a se stessi. Nella fattispecie, successivamente alle recenti modifiche del sito *CS Unplugged* [23], sono state introdotte all'interno di tre dei cinque argomenti totali (*Binary numbers*, *Error detection and correction* e *Kidbots*) delle *"Programming challenges"*. Esse consistono in esercizi di programmazione e di logica risolubili attraverso *Scratch* e, alcune, tramite *Python*. Le "challenges" permettono di ricreare le attività

unplugged, compresi i *"fogli di lavoro"* presenti in ognuna di esse, fornendone diverse varianti. Per ogni attività vengono anche mostrati quali blocchi di *Scratch* sono necessari per il loro svolgimento. Inoltre, negli argomenti *Binary numbers* ed *Error detection and correction*, nella sezione *Looking for more?*, è presente una versione interattiva dell'attività *unplugged*.

In *Code.org*, come precedentemente analizzato in 1.4.1, sono presenti delle attività *plugged*, il cui scopo è introdurre gli studenti alla programmazione, tramite un ambiente visuale a blocchi. Le attività sono composte da esercizi di programmazione e di logica nei quali, la maggior parte delle volte, viene chiesto di programmare i movimenti di un personaggio virtuale, spesso già protagonista di altri giochi esistenti, che deve svolgere delle particolari mansioni. È inoltre possibile vedere la "traduzione" del codice generato dai blocchi in codice *JavaScript*.

La conseguenza a questo tipo di approcci è la totale mancanza dell'apprendimento creativo, mostrato nella figura 3.1 sotto forma di una spirale, componente fondamentale del *pensiero computazionale*. Infatti fra gli elementi comuni delle principali definizioni di pensiero computazionale troviamo il *pensiero algoritmico e logico*, la *scomposizione di problemi*, l'*astrazione*, il *riconoscimento di pattern*, la *parallelizzazione*, la *valutazione*, la *programmazione*, il *tinkering*, il *debugging*, la *creazione* e la *collaborazione* che sono tutti elementi intrinseci dell'apprendimento creativo incentivato tramite la programmazione.

Durante lo svolgimento delle suddette attività i discenti nè *creano progetti*, nè modificano o "remixano" quelli già esistenti, ma svolgono solamente degli esercizi. Inoltre, essendo gli esercizi uguali per tutti non viene in alcun modo stimolata nè la loro condivisione, nè la passione e l'interesse propri degli studenti. Per di più i discenti non hanno la possibilità di giocare, in quanto ogni esercizio è fine a se stesso e non, per esempio, parte di un progetto più grande.

Al contrario, tramite lo svolgimento dei giochi per l'insegnamento dell'algoritmo del massimo (e del minimo), *gara di macchinine* e *sfida ai tiri liberi*

di basket, presentati in 2.3, si sono unite tutte le caratteristiche dell'apprendimento creativo. Inizialmente si insegna agli studenti a capire e a scrivere l'algoritmo del massimo necessario per il corretto funzionamento dei giochi, così da poterli effettivamente testare giocandoci (*play*); successivamente gli studenti possono vedere e modificare sia gli altri script, che gli aspetti dell'interfaccia grafica dei giochi, creando dei loro progetti (*projects*) personalizzati da condividere (*peers*). Inoltre grazie all'unione delle conoscenze pregresse e di quelle appena apprese, potranno ricreare gli stessi o altri contenuti digitali interattivi in base ai loro interessi e alle loro passioni (*passion*).

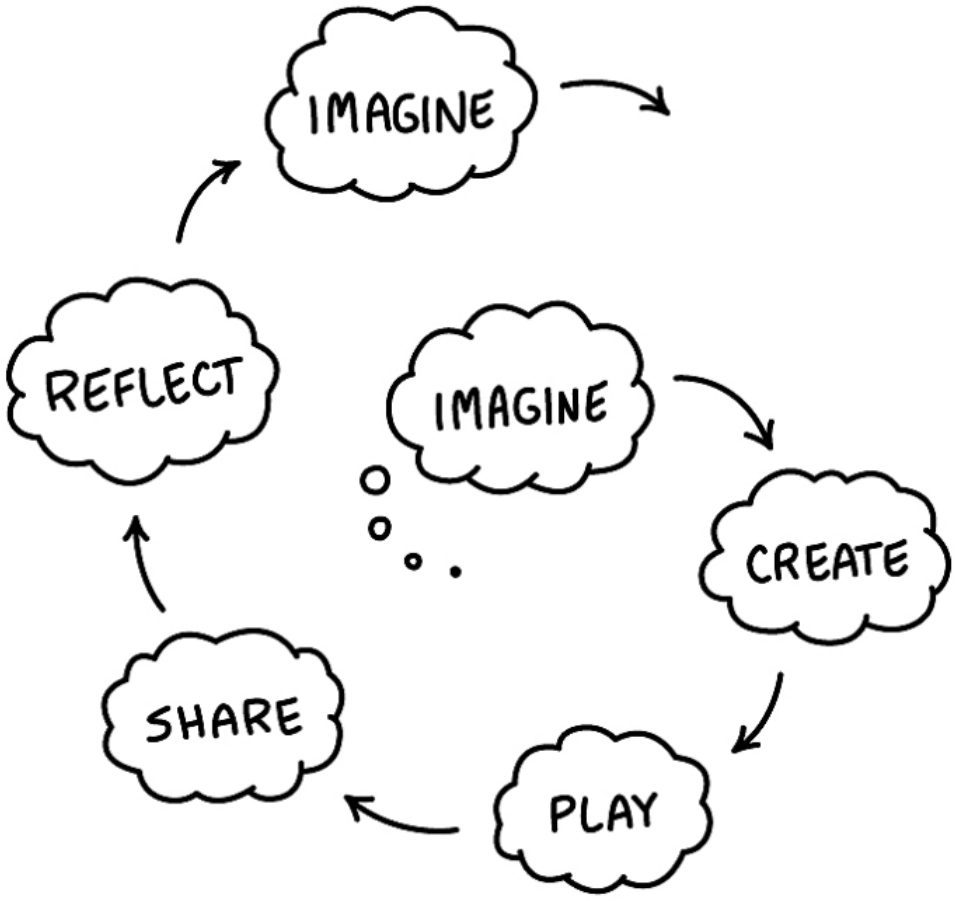

Figura 3.1: Spirale dell'apprendimento creativo [68]

3.2 Sviluppi futuri

Il passo immediatamente successivo allo sviluppo dei due giochi per calcolare il massimo, presentati e descritti in precedenza, è quello di provarli effettivamente sugli studenti a partire dal terzo anno della scuola primaria, in quanto al momento sono stati solamente discussi e analizzati con alcuni esperti di didattica dell'informatica. In particolare, sarebbe interessante analizzare come gli studenti reagiscono a questo tipo approccio e quali sono gli aspetti positivi e negativi evidenziati, in modo da riuscire a modificare i giochi in relazione a quanto osservato.

Successivamente a queste prove si potrebbero sviluppare altre attività quali giochi o storie che per essere utilizzate richiedano la costruzione da parte dei discenti di qualche algoritmo fondamentale dal punto di vista della didattica dell'informatica e della programmazione.

Un altro aspetto rilevante è quello di riuscire a sviluppare un *ambiente di programmazione visuale*, ispirato alla teoria costruzionista dell'apprendimento e progettato per l'insegnamento della programmazione, che integri le caratteristiche positive degli ambienti di programmazione precedentemente analizzati in 2.4, a discapito di quelle negative. Un'ottima base di partenza è *BloP* (2.4.2), al quale però si dovrebbero apportare alcune modifiche. In particolare, fra le sue peculiarità, quelle da modificare sono:

1. premendo sulla bandierina verde si inizia l'esecuzione dello script posizionato più in alto nella *Scripting Area* dello *sprite* selezionato;

2. non è possibile definire o rimuovere i blocchi personalizzati;

3. non è possibile nè modificare le impostazioni di Snap! nè utilizzare il salvataggio e la condivisione sul cloud;

4. avere l'obbligo di inserire i blocchi *quando si esegue uno script* ed *esegui altri script* ogniqualvolta si voglia eseguire un programma nel quale non siano stati precedentemente caricati dei blocchi di uno specifico linguaggio di programmazione.

La prima peculiarità andrebbe rimossa in quanto il fatto che gli script vengano eseguiti in base alla posizione in cui si trovano nello sprite selezionato, non permette più di avere una *programmazione parallela*. Questo comporta un aumento delle probabilità di incorrere in errori di programmazione concorrente, ad esempio *race condition*, in quanto si ha un cospicuo aumento di blocchi, in particolare, di quelli necessari per effettuare scambi di messaggi fra i vari sprite. Inoltre, come sostenuto in [45] ed analizzato in 2.4.1 è meglio insegnare ai nuovi studenti a programmare in parallelo piuttosto che in modo sequenziale.

Sarebbe inoltre interessante, soprattutto in ambito didattico, avere la possibilità di modificare i blocchi personalizzati all'interno di *BloP*. In questo modo l'insegnante potrebbe "sbloccare" l'interfaccia di *Snap!*, creare alcuni blocchi lasciando vuote le loro definizioni, decidere quali primitive mostrare e quali nascondere e infine "bloccare" l'interfaccia di *Snap!* passando a *BloP*. A questo punto, gli studenti si ritroverebbero a doversi concentrare nella costruzione dell'algoritmo da inserire nei blocchi vuoti, precedentemente creati, tramite le primitive a loro disposizione. Contemporaneamente l'insegnante avrà la certezza che non si distrarranno con tutti i vari dettagli grafici, in quanto la GUI è "bloccata".

Un'importante caratteristica da integrare riguarda la mancata possibilità di salvare sul cloud e condividere i propri progetti con altre persone. La condivisione e la "collaborazione creativa" [73] sono punti fondamentali in *Scratch*: la prima consente di visualizzare e provare i progetti di altre persone, modificarne il codice e crearne i "remix"; la seconda permette la connessione, collaborazione e coordinazione di diversi utenti nello sviluppo di progetti e attività.

BloP, come osservato in precedenza, è stato sviluppato per permettere agli insegnanti di creare delle versioni di qualsiasi linguaggio di programmazione tramite uno stile *a blocchi*. Per svolgere questa funzione i blocchi *quando si esegue uno script* ed *esegui altri script*, non sono obbligatori. È importante espandere la non obbligatorietà di questi due blocchi anche all'uso didattico

che ne viene fatto di BloP in questa tesi, così da semplificare il lavoro sia all'insegnante che allo studente.

Oltre alle precedenti modifiche risulterebbe utile, da un punto di vista didattico, l'inserimento del bottone *visible stepping button*, già presente in *Snap!*, che mostra l'esecuzione degli script istruzione per istruzione.

Conclusioni

> *L'obiettivo principale della scuola è quello di creare uomini che sono capaci di fare cose nuove, e non semplicemente ripetere quello che altre generazioni hanno fatto.*
>
> — Jean Piaget

L'obiettivo della tesi era quello di ideare e sviluppare una o più attività per l'insegnamento dell'algoritmo del massimo agli studenti fra il terzo e il quinto anno della scuola primaria, attraverso un approccio *costruttivista* che mettesse in risalto le caratteristiche del *pensiero computazionale*.

Alla fine del percorso, possiamo affermare di essere riusciti nel conseguimento del nostro obiettivo.

Siamo partiti dall'analisi delle diverse metodologie didattiche per lo sviluppo di attività *plugged* e *unplugged* atte all'insegnamento dell'informatica. Successivamente, dopo averne analizzate e descritte le caratteristiche positive e negative, abbiamo studiato ed esplorato i diversi *ambienti di sviluppo visuali* che vengono utilizzati per la didattica dell'informatica a livello mondiale, in quanto ognuno di essi presenta particolari caratteristiche didatticamente utili.

Con le conoscenze acquisite dagli studi precedenti siamo riusciti a sviluppare due differenti attività che hanno permesso il raggiungimento dell'obiettivo prefissato.

Spingendoci ancora oltre abbiamo generalizzato lo schema progettuale utilizzato per lo sviluppo delle suddette attività, cosicché, in futuro, possa essere riutilizzato sia per l'insegnamento di altri algoritmi, sia per la modifica di quelli esistenti adattandoli ad altre classi d'età.

In ultima battuta abbiamo ideato e descritto una possibile implementazione di un ambiente di sviluppo visuale a blocchi, di tipo *drag-and-drop*. Questo è stato possibile grazie agli studi effettuati e alle analisi derivate dall'utilizzo di diversi ambienti di programmazione visuali, utilizzati per lo sviluppo delle attività per l'insegnamento dell'algoritmo del massimo, estrapolandone le caratteristiche didatticamente utili di ognuno di essi.

Appendice A

Schema progettuale dei giochi

> *Nell'uomo autentico si nasconde un bambino: che vuole giocare.*
>
> — F. Nietzsche

Per l'ideazione e il conseguente sviluppo dei giochi per l'insegnamento dell'algoritmo del massimo (descritti in 2.3) si è utilizzato il medesimo schema architetturale. Tramite la generalizzazione di questo modello è possibile creare ulteriori attività interattive (come giochi, storie e arte) per l'insegnamento di altri algoritmi (ad esempio: *algoritmi per l'ordinamento*, *algoritmi per la ricerca* etc.) agli studenti. Per la sua composizione si è fatto riferimento a [32]:

- **Nome dell'attività**. Assegnare un nome che espliciti il tipo di attività interattiva che si andrà a svolgere e che attiri la curiosità dei discenti.
- **Scelta dell'algoritmo**. Scegliere quale algoritmo e quali costrutti di programmazione si intendono introdurre ed insegnare agli studenti.
- **Problema**. Spiegare quale problema dovrà essere risolto mediante l'utilizzo dell'algoritmo scelto.
- **Sviluppo**. Sviluppare l'attività strutturandola in modo che per funzionare richieda indispensabilmente la scrittura dell'algoritmo scelto.

- **Contesto**. Decidere in quale contesto o situazione l'attività sia praticabile. L'età e le conoscenze pregresse sono due elementi da tenere in forte considerazione quando si sceglie l'algoritmo e/o i costrutti di programmazione da insegnare e quando si sviluppa l'attività tramite la quale insegnarli.

- **Condizione**. Le condizioni in cui questo schema è applicabile. Si deve tenere in considerazione che per svolgere qualsiasi attività è essenziale l'utilizzo di ambienti di sviluppo visuali a blocchi e conseguentemente dei computer. Inoltre l'attività deve essere accessibile e fruibile da tutti gli insegnanti quindi si deve cercare di renderla il più facile e comprensibile possibile, cosicché risulti utile per diverse classi scolastiche.

- **Soluzione**. La strategia per risolvere il problema. L'attività deve contenere una sfida che stimoli e coinvolga gli studenti durante la ricerca di una soluzione. Essa sarà poi valutata in base ai criteri decretati dall'insegnante mostrando sia l'esistenza di soluzioni diverse per risolvere uno stesso problema sia l'esistenza di problemi diversi risolubili da soluzioni simili.

- **Risultato dell'utilizzo di questo modello**. Aumentare la quantità e la qualità delle attività interattive disponibili per l'insegnamento degli algoritmi e della programmazione, cosicché gli insegnanti abbiano un ventaglio sempre più ampio di attività fra cui scegliere.

Bibliografia

[1] Bee bot. https://www.bee-bot.us/.

[2] Black girls code. http://www.blackgirlscode.com/.

[3] Code.org. https://code.org/.

[4] Django girls. https://djangogirls.org/.

[5] Fondazione mondo digitale. http://www.mondodigitale.org/it.

[6] Girls code it better. http://www.girlscodeitbetter.it/.

[7] Girls who code. https://girlswhocode.com/.

[8] Ozo bot. https://ozobot.com/.

[9] Programma il futuro. https://programmailfuturo.it/.

[10] Rails girls. http://railsgirls.com/.

[11] I nativi digitali non esistono. http://tecnodidatticamente.blogspot.it/2013/02/i-nativi-digitali-non-esistono.html, 2013.

[12] National curriculum in england: computing programmes of study. https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study, settembre 2013.

[13] A Balanskat and K Engelhardt. Computer programming and coding: Priorities, school curricula and initiatives across europe, european schoolnet, 2015. European Schoolnet (EUN Partnership AIBSL).

[14] Filippo Bartolini. Gara di macchinine. https://scratch.mit.edu/projects/191294401/, dicembre 2017.

[15] Filippo Bartolini. Gara di macchinine. https://tinyurl.com/GaraDiMacchine, dicembre 2017.

[16] Filippo Bartolini. Gara di macchinine (versione 2). https://scratch.mit.edu/projects/194275690/, dicembre 2017.

[17] Filippo Bartolini. Gara di macchinine (versione 2). https://tinyurl.com/GaraDiMacchineVersione2, gennaio 2018.

[18] Filippo Bartolini. Gara di macchinine (versione 2). https://scratch.mit.edu/projects/embed-editor/201862381/?isMicroworld=true, gennaio 2018.

[19] Filippo Bartolini. Sfida ai tiri liberi a basket. https://scratch.mit.edu/projects/200276699/, gennaio 2018.

[20] Filippo Bartolini. Sfida ai tiri liberi a basket. https://tinyurl.com/SfidaAiTiriLiberiABasket, gennaio 2018.

[21] Filippo Bartolini. Sfida ai tiri liberi a basket. https://scratch.mit.edu/projects/embed-editor/201865342/?isMicroworld=true, gennaio 2018.

[22] Tim Bell. Computer science unplugged. http://csunplugged.org/activities/.

[23] Tim Bell. Computer science unplugged. https://csunplugged.org/en/topics/.

[24] Carla Bertacchini. *Educazione e tecnologie: i nuovi strumenti della mediazione didattica*. Junior, 2002.

[25] Giovanni M Bianco. *L'informatica è un gioco. Tecniche di mediazione per la formazione primaria*. Giovanni M. Bianco, 2014.

[26] Chiara Bodei and Linda Pagli. L'informatica: non è un paese per donne. *Mondo Digitale*, page 2, 2017.

[27] Rosa Maria Bottino. Ict as a catalyst of innovation. In *ICT in Education in Global Context*, pages 3–18. Springer, 2014.

[28] Alessia Cadamuro and Alessandra Farneti. *Insegnanti e bambini: Idee e strumenti per migliorare la relazione*. Carocci, 2008.

[29] V. Campione. *La didattica nell'era digitale*. Quaderni di Astrid. Il Mulino, 2015.

[30] Tonino Cantelmi. L' era digitale e la sua valenza antropologica: i nativi digitali. *Relazione presentata al III Convegno Internazionale della Società Italiana di Psicotecnologie e Clinica dei nuovi Media–SIP tech–Palermo*, 2009.

[31] Lillian Cassel, Alan Clements, Gordon Davies, Mark Guzdial, Renée McCauley, Andrew McGettrick, Bob Sloan, Larry Snyder, Paul Tymann, and Bruce W. Weide. Computer science curriculum 2008: An interim revision of cs 2001. Technical report, New York, NY, USA, 2008.

[32] James O Coplien and A Word On Alexander. Software patterns. 1996.

[33] Isabella Corradini, Michael Lodi, and Enrico Nardelli. Computational thinking in italian schools: Quantitative data and teachers' sentiment analysis after two years of "programma il futuro". In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, ITiCSE '17, pages 224–229, New York, NY, USA, 2017. ACM.

[34] Lifelong Kindergarten Group dei Media Lab del MIT. Scratch. https://scratch.mit.edu/.

[35] Lifelong Kindergarten Group dei Media Lab del MIT. Scratch microworlds. https://scratch.mit.edu/microworlds/go, 2017.

[36] Marco Durante. I nativi digitali. http://www.labcd.unipi.it/wp-content/uploads/2015/05/MarcoDurante_Natividigitali.pdf, 2015.

[37] Stefano Federici. Blop - build your own block language. https://sites.google.com/site/blocklanguages/home, 2013.

[38] G Fiorentino. *Tecnologie per educare.* Carocci Editore, 2013.

[39] Benjamin R Harris and Michelle S Millet. Nothing to lose: "fluency" in information literacy theory and practice. *Reference services review*, 34(4):520–535, 2006.

[40] Brian Harvey and Jens Mönig. Snap! (build your own blocks). http://snap.berkeley.edu/, 2015.

[41] Brian Harvey and Jens Mönig. Snap! reference manual. *URL http://snap. berkeley. edu/SnapManual. pdf*, 2017.

[42] Felienne Hermans and Efthimia Aivaloglou. To scratch or not to scratch?: A controlled experiment comparing plugged first and unplugged first programming lessons.

[43] Johan Huizinga. *Homo Ludens Ils 86*, volume 3. Routledge, 2014.

[44] Association for Computing Machinery (ACM) Joint Task Force on Computing Curricula and IEEE Computer Society. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. ACM, New York, NY, USA, 2013. 999133.

[45] Keith Kirkpatrick. Parallel computational thinking. *Communications of the ACM*, 60(12):17–19, 2017.

[46] MIT Media Lab. Creative learning. http://learn.media.mit.edu/creative-learning.

[47] Raffaello Lambruschini. *Della educazione e dell'istruzione*. GP Viesseux, 1850.

[48] Marcia C. Linn and Bat-Sheva Eylon. *Science Education: Integrating Views of Learning and Instruction*, pages 511–544. Lawrence Erlbaum Associates, Mahwah, NJ, 2006.

[49] L Liukas. Linda liukas: A delightful way to teach kids about computers [video file]. *Retrieved from https://www. ted. com/talks/linda_liukas_a_delightful_way_to_teach_kids_about_computers/up-next*, 2015.

[50] L. Liukas. *HELLO RUBY: Avventure nel mondo del coding*. STEM. Science, Technology, Engineering, Mathematics. Erickson, 2017.

[51] Michael Lodi. Imparare il pensiero computazionale, imparare a programmare. Master's thesis, Università di Bologna, Corso di Studio in Informatica, 2014. http://amslaurea.unibo.it/6730/.

[52] Michael Lodi, Simone Martini, and Enrico Nardelli. Abbiamo davvero bisogno del pensiero computazionale? *Mondo Digitale*, 2017.

[53] Hansel Lynn. The coder school. https://www.thecoderschool.com/.

[54] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):16, 2010.

[55] Orni Meerbaum-Salant, Michal Armoni, and Mordechai Ben-Ari. Habits of programming in scratch. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*, pages 168–172. ACM, 2011.

[56] Microsoft. Why don't european girls like science or technology? https://news.microsoft.com/europe/features/dont-european-girls-like-science-technology/.

[57] MIUR. Animatore digitale. http://hubmiur.pubblica.istruzione.it/web/ministero/cs191115, novembre 2015.

[58] Katia Montalbetti. *La pedagogia sperimentale di Raymond Buyse: ricerca educativa tra orientamenti culturali e attese sociali*. Vita e Pensiero, 2002.

[59] Tomohiro Nishida, Susumu Kanemune, Yukio Idosaka, Mitaro Namiki, Tim Bell, and Yasushi Kuno. A cs unplugged design pattern. In *ACM SIGCSE Bulletin*, volume 41, pages 231–235. ACM, 2009.

[60] OECD. Students, computers and learning: Making the connection. http://dx.doi.org/10.1787/9789264239555-en, 2015.

[61] Seymour Papert. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980.

[62] Seymour Papert. *Constructionism: A new opportunity for elementary science education*. Massachusetts Institute of Technology, Media Laboratory, Epistemology and Learning Group, 1986.

[63] David Perkins. *Smart schools: better thinking and learning for every child*. New York, 1995.

[64] Sara Piccolo and Francesca Puliti. L'informatica non è roba da ragazze? negli anni 70-80 lo era, eccome. *La Repubblica*, dicembre 2015.

[65] Marc Prensky. Digital natives, digital immigrants part 1. *On the horizon*, 9(5):1–6, 2001.

[66] Emanuele Rapetti and Lorenzo Cantoni. "nativi digitali" e apprendimento con le ict. *Journal of e-learning and knowledge society*, 6(1):43–53, 2010.

[67] M Resnick, M Ito, U Gasser, N Rusk, and P Schmidt. Coding for all: interest-driven trajectories to computational fluency. *Proposal to the National Science Foundation*, 2013.

[68] Mitchel Resnick. All i really need to know (about creative thinking) i learned (by studying how children learn) in kindergarten. In *Proceedings of the 6th ACM SIGCHI conference on Creativity & cognition*, pages 1–6. ACM, 2007.

[69] Mitchel Resnick. Give p's a chance: Projects, peers, passion, play. In *Constructionism and creativity: Proceedings of the Third International Constructionism Conference. Austrian Computer Society, Vienna*, pages 13–20, 2014.

[70] Mitchel Resnick and D Siegel. A different approach to coding. *Bright/Medium*, 2015.

[71] K Robinson. Ken robinson: Do schools kill creativity?[video file]. *Retreived from http://www. ted. com/talks/ken_robinson_says_schools_kill_creativity*, 2006.

[72] K Robinson. Ken robinson: How to escape education's death valley [video file]. *Retrieved from https://www. ted. com/talks/ken_robinson_how_to_escape_education_s_death_valley*, 2013.

[73] Ricarose Roque, Natalie Rusk, and Mitchel Resnick. Supporting diverse and creative collaboration in the scratch online community. In *Mass collaboration and education*, pages 241–256. Springer, 2016.

[74] Reshma Saujani. Teach girls bravery, not perfection. *TED Talks (New York, NY)*, 2016.

[75] Daniel L Schwartz and John D Bransford. A time for telling. *Cognition and instruction*, 16(4):475–5223, 1998.

[76] Rivka Taub, Michal Armoni, and Mordechai Ben-Ari. Cs unplugged and middle-school students’ views, attitudes, and intentions regarding cs. *Trans. Comput. Educ.*, 12(2):8:1–8:29, April 2012.

[77] Moran Tsur. *Scratch Microworlds: introducing novices to scratch using an interest-based, open-ended, scaffolded experience*. PhD thesis, Massachusetts Institute of Technology, 2017.

[78] Sherry Turkle and Seymour Papert. Epistemological pluralism and the revaluation of the concrete. *Journal of Mathematical Behavior*, 11(1):3–33, 1992.

[79] Lev S Vygotskij. Il ruolo del gioco nello sviluppo mentale del bambino. *Bruner, Jerome S., Jolly, Alison e Sylva Kathy (a cura di), Il gioco: ruolo e sviluppo del comportamento ludico negli animali e nell'uomo, Roma, Armando*, 4:657–678, 1981.

[80] Jeannette M Wing. Computational thinking. *Communications of the ACM*, 49(3):33–35, 2006.

Ringraziamenti

Ringrazio il Professor Renzo Davoli, per avermi dato l'oppurtunità di lavorare con lui ad un progetto, per me, particolarmente interessante e significativo, guidandomi con pazienza e professionalità alla stesura di questa tesi.

Ringrazio il Dottor Michael Lodi, per avermi seguito e consigliato durante tutta la fase di progettazione e sviluppo della tesi. Vedo in lui un ottimo futuro docente.

Ringrazio i miei genitori, perché senza i loro sacrifici non avrei potuto raggiungere questo importante obiettivo.

Ringrazio i parenti che mi hanno supportato durante questa mia esperienza universitaria.

Ringrazio la mia fidanzata Sonia che mi è sempre stata vicina, anche nei momenti di difficoltà, spronandomi ed incoraggiandomi a dare il meglio di me e che spesso mi ha ospitato nella bellissima Bologna durante i miei anni di studio da pendolare.

Ringrazio tutti i miei amici di Rimini con i quali ho condiviso e condividerò le gioie e i traguardi della mia vita.

Ringrazio gli amici e i colleghi universitari di Bologna che, in modi diversi, mi hanno accompagnato durante questo bellissimo e tortuoso percorso.