Soluzione Progetto 2 ASD a.a. 2018/2019

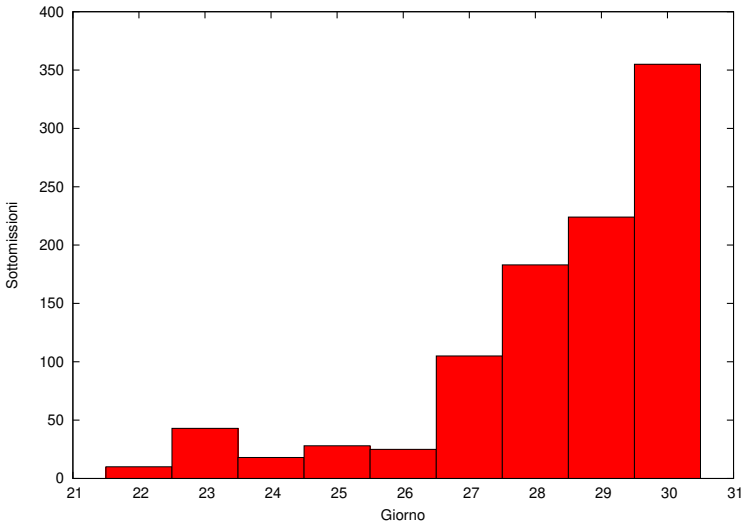
Game of (Approximated) Thrones

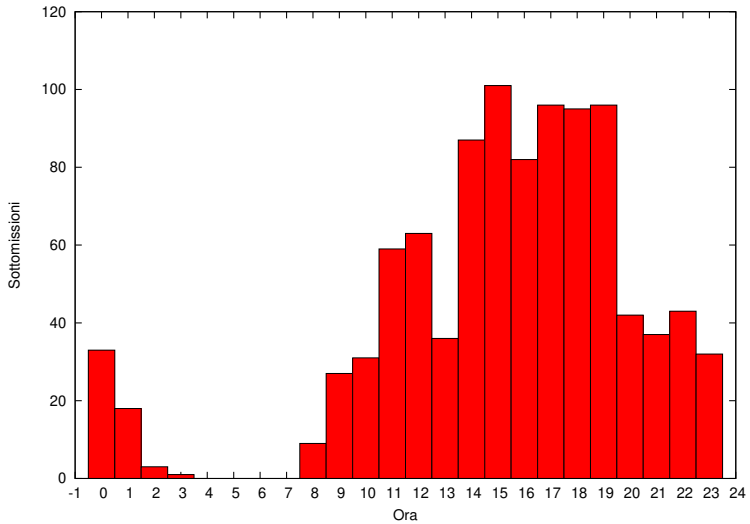
Cristian Consonni e Marta Fornasier

03 giugno 2019

Statistiche

Numero sottoposizioni: 991





- 88 gruppi partecipanti, di cui 58 gruppi hanno fatto almeno una sottoposizione;
- 185 studenti iscritti, di cui 132 appartenenti a gruppi che hanno fatto almeno una sottoposizione;
- 10 ore di ricevimento (compresi i laboratori);

Risultati

Punteggi (classifica completa sul sito)

- $P < 40$ $\longrightarrow$ progetto non passato
- $40 \leq P < 59$ $\longrightarrow$ 1 punto bonus (13 gruppi, 29 studenti)
- $59 \leq P < 75$ $\longrightarrow$ 2 punti bonus (17 gruppi, 37 studenti)
- $P \geq 75$ $\longrightarrow$ 3 punti bonus (17 gruppi, 45 studenti)

`https://judge.science.unitn.it/slides/asd18/classifica_prog2.pdf`

Il problema

Fillomino

Il progetto è ispirato ad un problema conosciuto come Fillomino[1]. Fillomino è un puzzle della stessa casa editrice che ha reso famoso il Sudoku. Nella versione decisionale è un problema NP-COMPLETE[2].

- Non è quindi possibile individuare la soluzione ottima in tempo polinomiale (se $\mathrm{P} \neq \mathrm{NP}$).
- Per affrontare il problema esistono diversi approcci euristici.

[1] `https://en.wikipedia.org/wiki/Fillomino`

[2] `http://bit.ly/fillomino-NPc`

Osservazioni

- Per affrontare questo problema è conveniente vedere la mappa come un **grafo** (non orientato). In questo grafo, i nodi sono le singole celle e gli archi collegano ogni nodo con i nodi adiacenti (sopra, sotto, a destra e a sinistra).
- Qualsiasi sia l'approccio, saranno utili gli algoritmi di dfs e bfs per visitare il grafo. Questi algoritmi possono essere applicati, ad esempio, per la ricerca delle componenti connesse.

Primo algoritmo greedy (I)

- Osserviamo che, per come è definito il punteggio, è importante massimizzare il **numero** di castelli utilizzati. Non contano i valori del castelli correttamente posizionati nelle varie suddivisioni.
- Per questo, un primo approccio greedy prevede di iniziare a creare suddivisioni partendo dai castelli con valore **minore**. Infatti, questi occuperanno meno spazio e renderanno meno difficile l'espansione degli altri castelli.
- Senza preoccuparci subito di creare suddivisioni valide, partendo da ogni castello, possiamo creare una suddivisione che lo contenga. Facciamo una visita della griglia ed ogni volta che troviamo una cella libera (con uno 0) ci scriviamo il valore del castello, fino a che (se possibile) raggiungiamo la dimensione desiderata.

$\Rightarrow$ Con questo approccio potrebbero crearsi suddivisioni adiacenti dello stesso valore o potrebbero esserci suddivisioni incomplete! Vediamo come possiamo risolvere questo problema nel modo più semplice e brutale possibile.

Primo algoritmo greedy (II)

Supponiamo di aver creato una griglia con alcune suddivisioni, anche non valide, ma **senza aver sovrascritto nessun castello**. Potrebbero quindi esserci territori di dimensione sbagliata.

$\Rightarrow$ C'è un metodo molto semplice per rendere **valida** la griglia.

- Possiamo vedere ora la mappa come un grafo i cui archi collegano solamente le celle adiacenti con lo stesso valore.
- Le suddivisioni create saranno le **componenti connesse** del grafo.
- Basterà allora calcolare la dimensione delle componenti connesse (con una dfs) ed individuare quelle la cui dimensione non corrisponde al valore scritto in ogni loro cella.
- Dopo averle individuate, con una seconda visita, potremo eliminare del tutto queste suddivisioni mettendo a 0 ogni loro cella.

Primo algoritmo greedy (III)

L'algoritmo che abbiamo descritto, riassumendo, procede così:

- effettua una visita per ogni castello (partendo da quelli più piccoli) e posiziona (come capita) suddivisoni;
- alla fine rende valida la griglia eliminando le suddivisioni di dimensione sbagliata.

$\Rightarrow$ Punteggio (circa): 40/100.

Primo algoritmo greedy (esempio) (I)

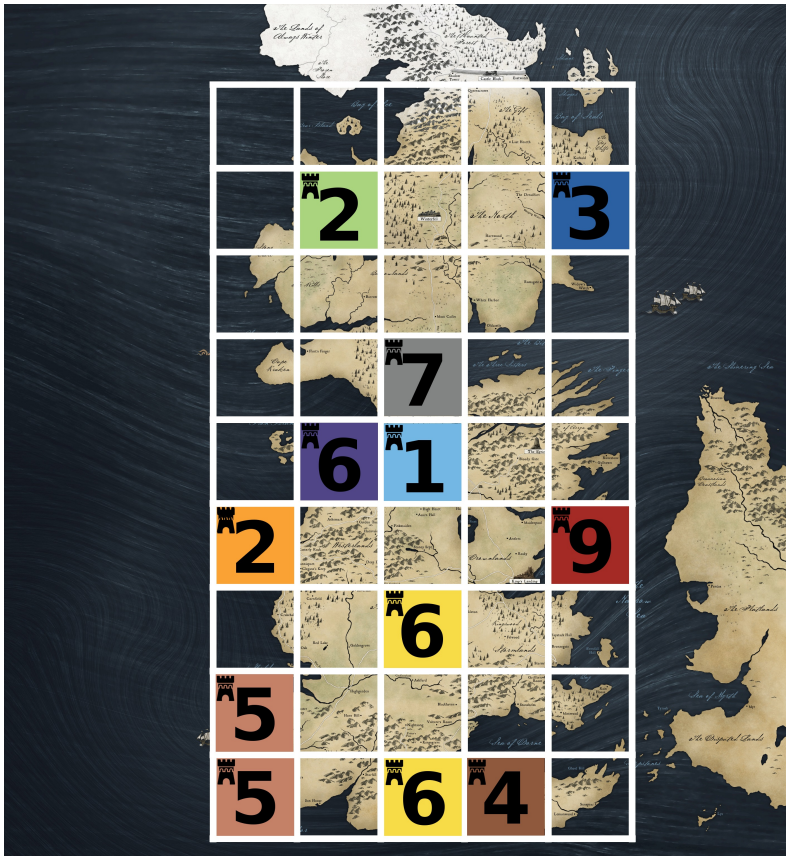


Figure: Mappa di input.

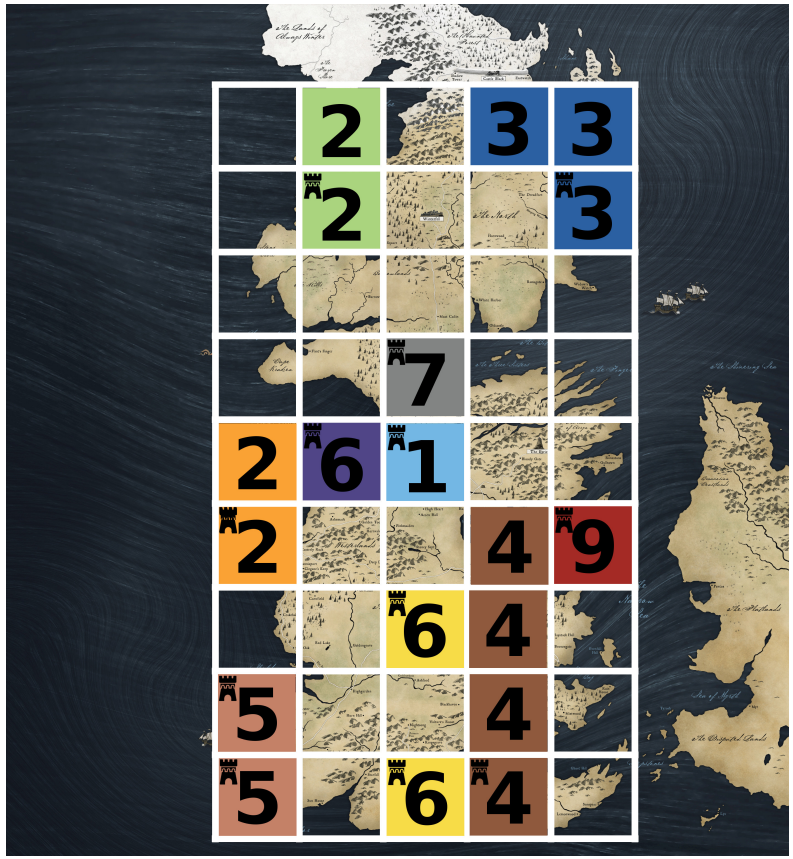


Figure: Vengono create le suddivisioni per i castelli più piccoli.

Primo algoritmo greedy (esempio) (II)

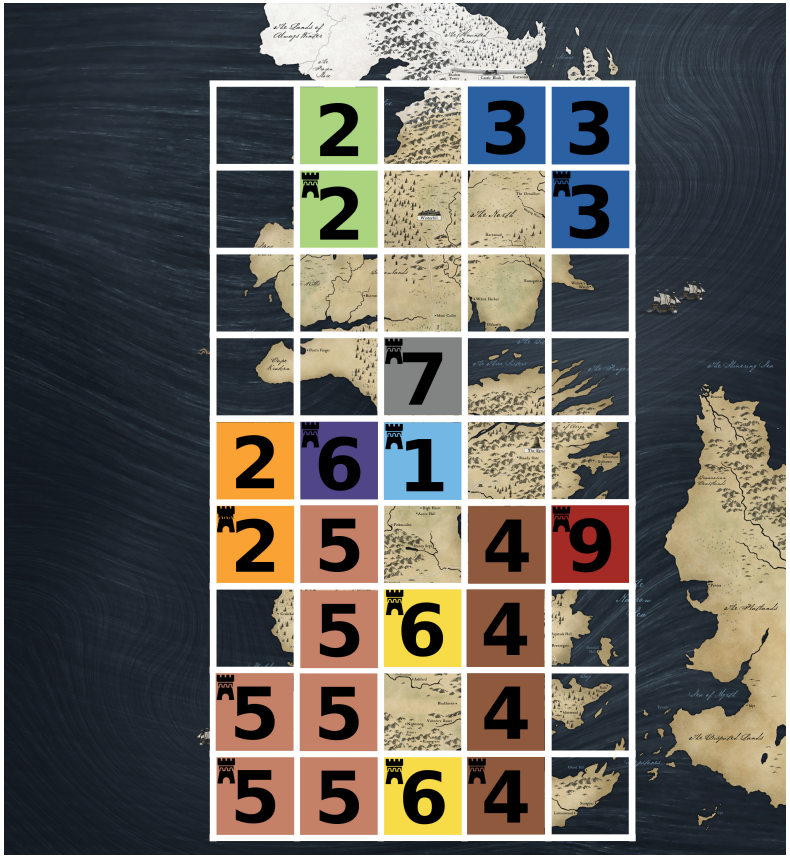


Figure: Viene creata una suddivisione per il primo castello di valore 5.

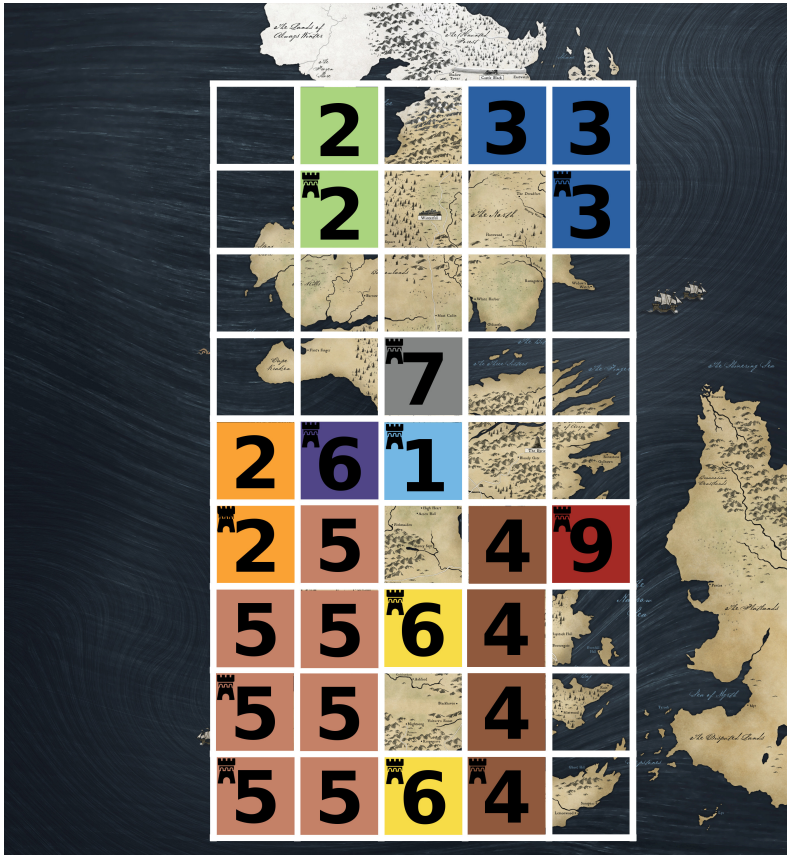


Figure: Si tenta di creare una suddivisione per il secondo castello di valore 5.

Primo algoritmo greedy (esempio) (III)

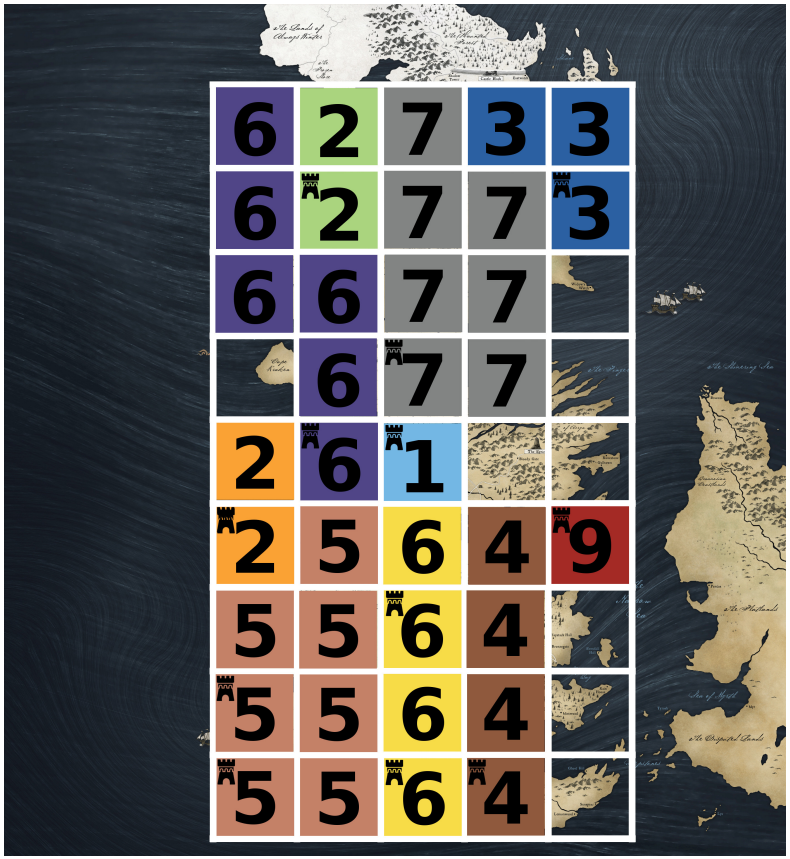


Figure: Vengono create suddivisioni per i castelli più grandi.

Figure: Alcune suddivisioni risultano incomplete.

Primo algoritmo greedy (esempio) (IV)

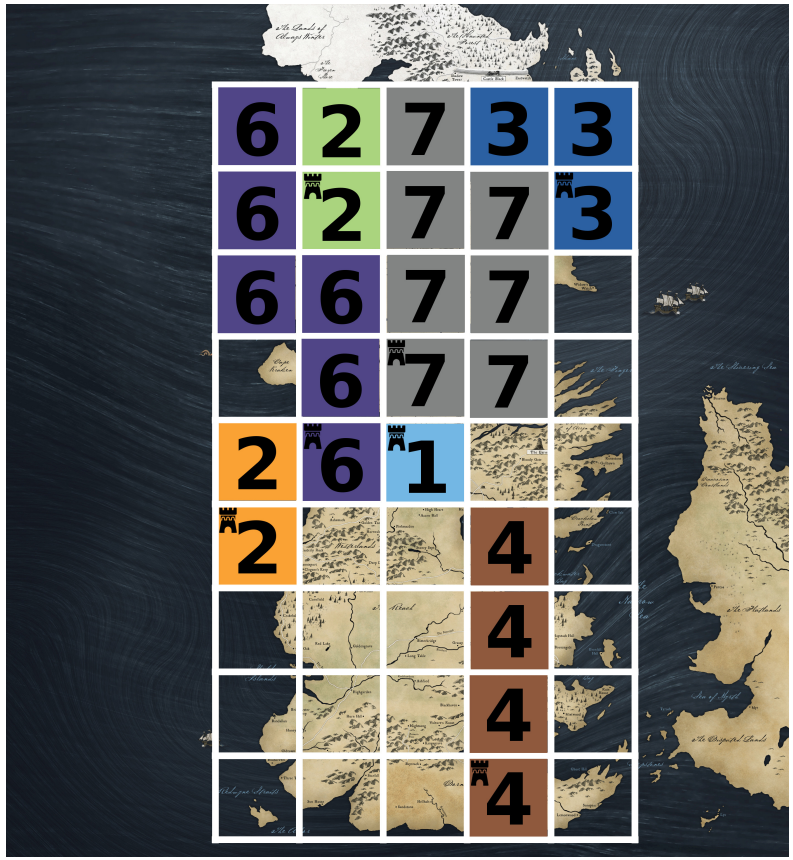


Figure: Vengono eliminate tutte le suddivisioni non valide.

Secondo algoritmo greedy (I)

Possiamo sicuramente fare di meglio! Iniziamo sopratutto a considerare la possibilità di unire più castelli in una stessa suddivisione, senza creare suddivisioni non valide (come in precedenza).

- Utilizziamo una **mappa di supporto** per la stampa in output.
- In questa mappa inseriremo una suddivisione solo dopo averla completata e dopo aver verificato essere valida.
- Da ogni castello partirà una visita della griglia, durante la quale ci sposteremo solo su celle **valide**, in ordine casuale. Spiegheremo in seguito come verificare la validità di una cella.

Secondo algoritmo greedy (II)

Ogni cella attraversata verrà memorizzata man mano in un vettore dinamico.

1. Se il vettore avrà raggiunto dimensione corrispondente a quella richiesta dal castello, scriveremo la suddivisione nella mappa di output.
2. In caso contrario, segnamo tutte le celle attraversate durante la visita come non visitate, rendendole di nuovo disponibili per i castelli successivi.

$\Rightarrow$ Se non avremo trovato una suddivisione valida la mappa di output non sarà modificata.

Secondo algoritmo greedy (III)

Le celle valide

Se il castello considerato è di valore v, avremo che una cella incontrata durante la visita è valida se rispetta le seguenti condizioni.

- Nessuna delle celle confinanti, nella mappa output, ha lo stesso valore v.
- La cella contiene uno 0 nella mappa originale e uno 0 nella mappa di output **oppure** contiene lo stesso valore v nella mappa originale e uno 0 nella mappa di output (in questo caso stiamo unendo due castelli!).
- La cella non è stata visitata.

Con questo approccio riusciamo a gestire bene la creazione di suddivisioni valide che possono comprendere più castelli.

$\Rightarrow$ Punteggio (circa): 70/100.

Secondo algoritmo greedy (esempi)

Vediamo come il secondo algoritmo greedy evita la creazione di suddivisioni non valide.

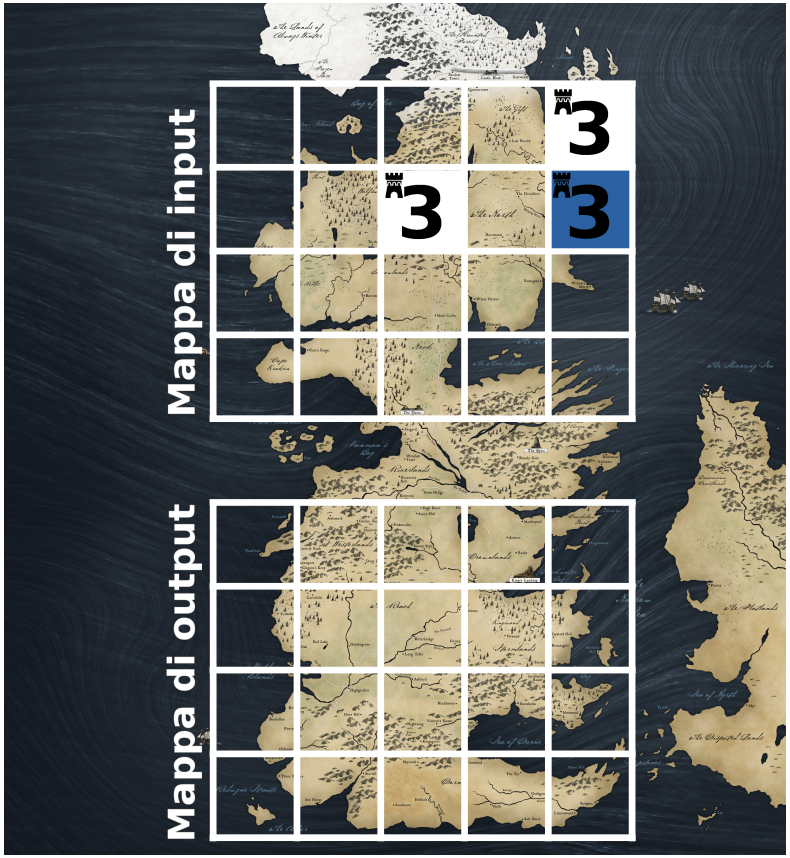


Figure: La visita parte dal castello azzurro di valore 3.

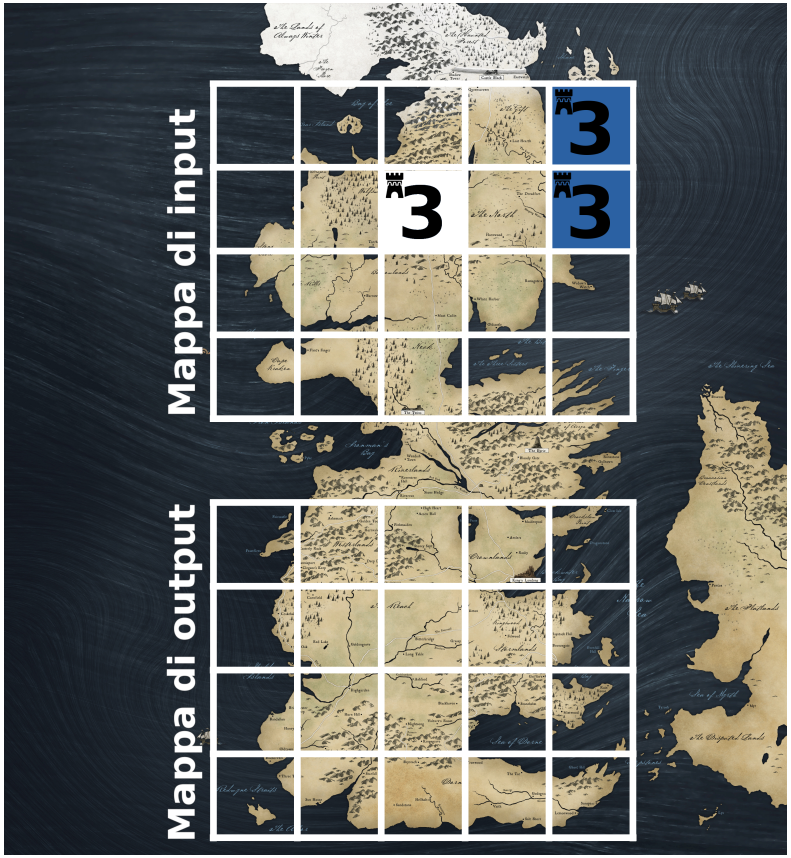


Figure: Procede verso l'alto incontrando un castello dello stesso valore.

Secondo algoritmo greedy (esempi)

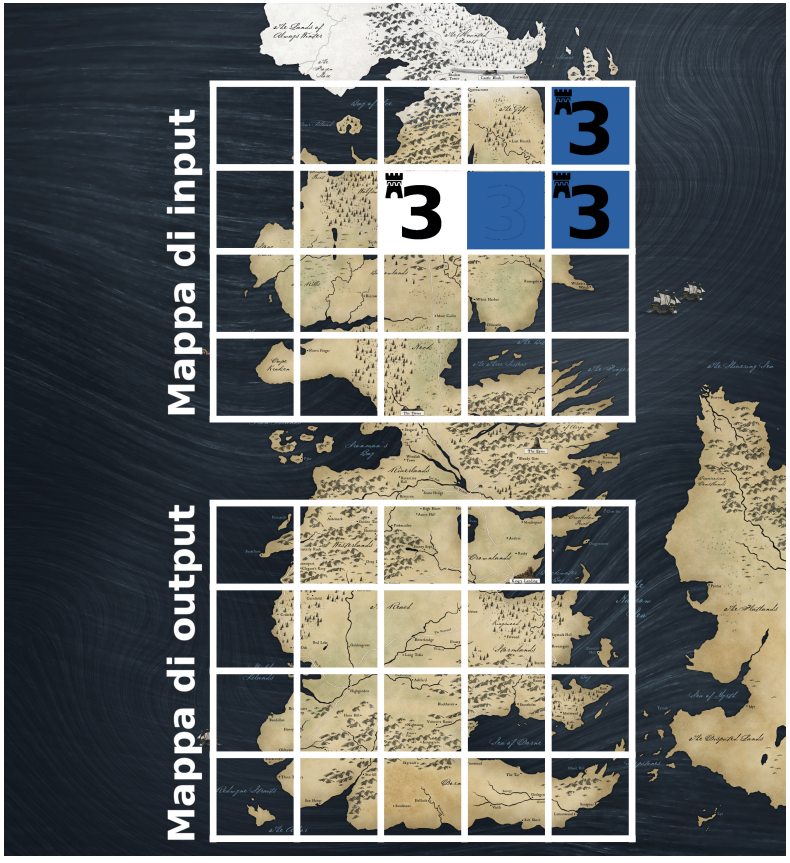


Figure: Procede poi a destra del castello considerato.

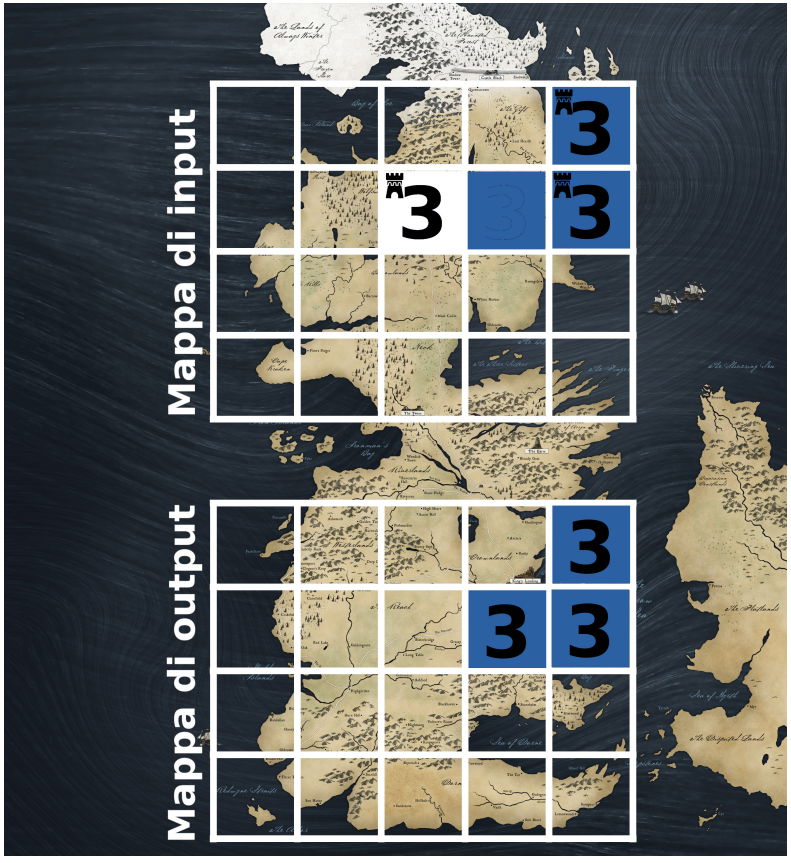
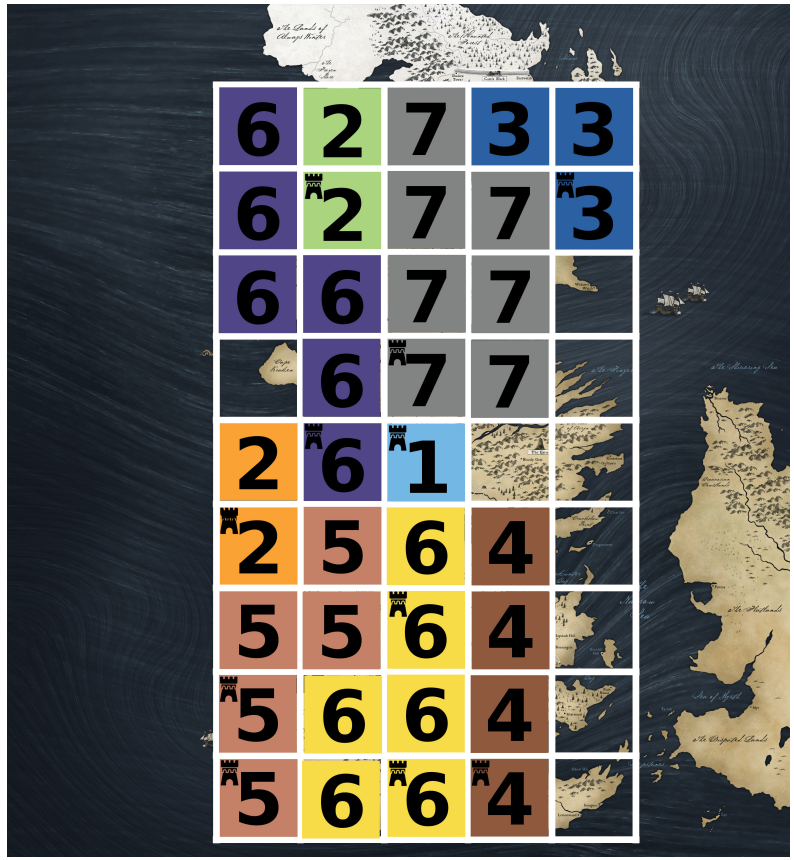


Figure: Infine scrive nella mappa di output la suddivisione trovata.

Notiamo che quando verrà visitato il castello bianco con valore 3, questo si troverà in una posizione non valida. Per questo non verrà mai inserito nella mappa di output in nessuna suddivisione.

Secondo algoritmo greedy (esempi)

Con il secondo algoritmo greedy, per quel che riguarda l'esempio visto in precedenza, la griglia viene riempita nel seguente modo.



Terzo algoritmo greedy (I)

L'ultimo algoritmo greedy che proponiamo segue l'idea secondo cui è meglio muoversi verso castelli dello stesso valore non ancora utlizzati ed è meglio muoversi lontano da castelli di valore diverso non ancora utilizzati. Questo approccio favorisce l'unione di castelli dello stesso valore in un'unica suddivisione, aumentando così il numero di spazi liberi che vengono lasciati agli altri castelli. Inoltre cerca di lasciare spazio ai castelli ancora non esaminati.
Per fare ciò, partendo da ogni castello, invece di muoverci casualmente su celle della mappa valide, calcoliamo una sorta di **precedenza** per le direzioni in cui muoverci.

Terzo algoritmo greedy (II)

Definiamo una funzione d per esprimere questa precedenza. Durante una visita che parte da un castello di valore v, la funzione d associa ad ogni cella valida, adiacente a quella considerata, un valore compreso tra 0 e 3. Le celle con valore d più basso saranno le prime esplorate.

- Se nella cella c'è un castello dello stesso valore, ma non assegnato, $d = 0$.
- Se ci sono celle adiacenti con un castello dello stesso valore, ma non assegnato, $d = 1$.
- Se ci sono celle adiacenti con castelli non assegnati di valore diverso da quello considerato, $d = 3$.
- Altrimenti, $d = 2$.

Le celle adiacenti verranno visitate in ordine di precedenza (dal valore d più basso al più alto).

Terzo algoritmo greedy (III)

Osservazione

In generale ogni algoritmo può essere applicato più volte, modificando l'ordine in cui vengono considerati i castelli. Ogni volta può essere calcolato il punteggio della soluzione ricavata e, se maggiore dei punteggi precedenti, la soluzione può essere nuovamente stampata.

Usando l'ultimo algoritmo greedy e stfruttando questa osservazione, applicando cioè l'algoritmo più volte, si ottiene un risultato abbastanza buono.

$\Rightarrow$ Punteggio (circa): 90/100.

Soluzione più avanzata (I)

Vi proponiamo infine la soluzione dei vostri colleghi, che raggiunge quasi il massimo dei punti. Si sviluppa essenzialmente in tre fasi.

- Nella prima fase si cerca di associare, in base alla dimensione e alla distanza (utilizzando una coda di priorità), i castelli con la stessa dimensione. Per fare ciò, si cerca il percorso più breve tra due regioni prese in considerazione (che inizialmente consisteranno in due castelli). Dopo aver associato due castelli in un'unica regione, per calcolare la distanza tra due nuove regioni bisognerà tener conto delle caselle già utilizzate. Le regioni verranno poi completate nelle fasi successive.
- L'idea di questa fase è completare le regioni muovendosi verso l'esterno (è probabile che verso l'esterno della matrice ci siano meno regioni in conflitto tra loro). Vengono così lasciati degli spazi liberi prevalementemente al centro della griglia. Durante questa fase le regioni incomplete vengono (momentaneamente) eliminate, per fare spazio al resto. Verranno considerate di nuovo nell'ultima fase.

Soluzione più avanzata (II)

- Questa fase crea lo spazio necessario per completare le regioni ancora incomplete, dando loro una cella alla volta. Per fare ciò cerca di "rubare" caselle dalle regioni adiacenti, senza rubare punti di articolazione (ovvero quelli che disconnettono la regione). Ogni cella della griglia sarà caratterizzata da un parametro, che indicherà se è un punto di articolazione o meno per la suddivisione di cui fa parte. Questo parametro viene calcolato con una dfs. Nel frattempo viene creato un grafo orientato che collega una regione ad un'altra se e solo se la prima può prendere una casella dalla seconda (cioè solo se si affaccia su punti di non articolazione). Attraverso questo grafo si calcola un percorso (con una bfs) dalla regione considerata ad una regione di spazi. Poi con una seconda visita si identificano quali caselle di ogni regione dare e prendere. In un caso particolare questa ricerca fallisce, e bisogna fare nuovi tentativi. Osserviamo come ad ogni passaggio è necessario ricalcolare i punti di articolazione e ricercare le caselle di interfacciamento.

Soluzione più avanzata (III)

L'aspetto importante di questa soluzione è che riesce a spostare le celle rimaste libere all'interno della mappa, per completare nuove suddivisioni.

$\Rightarrow$ Punteggio (circa): 99/100.

Altri approcci

- **Backtrack**: Alcuni di voi hanno implementato soluzioni in backtrack.
- **Algoritmi genetici**: Se qualcuno volesse, potrebbero rivelarsi una buona possibilità per scrivere un'altra soluzione. L'idea è la seguente. Gli algoritmi generitici considerano le soluzioni come cromosomi e simulano un processo evolutivo, ad esempio: i) incrociando soluzioni (*crossover*), ii) generando *mutazioni*. Una opportuna funzione di *selezione naturale* guida l'evoluzione dei cromosomi verso una soluzione ottima.