ChatGPT

Guida all'integrazione di Quackle 1.0.4 (Scrabble Engine) con DAWG e GADDAG

1. Architettura di Quackle e componenti principali

Struttura dei moduli: Quackle è organizzato in moduli distinti che separano il core logico dall'input/output e dagli strumenti di costruzione dei dizionari. Il **core engine** è la libreria `libquackle` (tipicamente compilata come libreria statica `liblibquackle.a`), che contiene tutta la logica di gioco (generazione delle mosse, regole, AI) ed è scritta in C++ puro senza dipendenze da Qt [1]. Sopra il core c'è la libreria **Quackle I/O** (`quackleio`), che gestisce funzioni di input/output: accesso ai dizionari (caricamento di file DAWG/GADDAG), serializzazione di partite (formato GCG), parsing di opzioni CLI, ecc. Questa parte ha alcune dipendenze da Qt (per utilità di stringa, file, ecc.) [2] [3].

Oltre alle librerie, Quackle fornisce **eseguibili standalone** per la generazione dei dizionari: in particolare `makeminidawg` (per costruire file lessicali compressi in formato DAWG) e `makegaddag` (per costruire file in formato GADDAG) [4]. Questi programmi console si appoggiano sul core engine (e in parte su quackleio) per prendere una wordlist di input e produrre il file lessicale binario corrispondente. Infine, sotto `quackle/data/` risiedono i **dati** di supporto: dizionari predefiniti (.dawg/.gaddag), strategie e soprattutto i file di **alfabeto** (.quackle_alphabet) per le varie lingue [4].

DAWG vs GADDAG – differenze e utilizzo: Quackle supporta due strutture dati per rappresentare il lexicon (insieme delle parole valide): il **DAWG** (Directed Acyclic Word Graph) e il **GADDAG**. Il DAWG è un grafo aciclico che compatta in modo efficiente tutte le parole condividendo prefissi comuni, risultando in file più piccoli. Il GADDAG, introdotto da Steven Gordon (1994) come evoluzione per la generazione rapida di mosse in Scrabble, memorizza per ogni parola anche i **prefissi rovesciati** attorno a ogni possibile "hook" (lettera di aggancio) [5]. Questo comporta che ogni parola venga memorizzata tante volte quanti sono i suoi prefissi, aumentando la dimensione del lessico di circa 5 volte rispetto a un DAWG semplice [6]. Il vantaggio è prestazionale: le ricerche di mosse con GADDAG sono circa **2× più veloci** degli algoritmi tradizionali basati su DAWG [7]. In pratica Quackle utilizza il GADDAG per generare tutte le possibili giocate (tenendo conto degli incastri con le lettere già sul tabellone), mentre il DAWG può essere usato per verificare rapidamente la validità di parole o per funzioni di completamento. In Quackle dalla versione 0.97 in poi, il motore di generazione mosse è passato ad usare **solo GADDAG** per massimizzare le prestazioni di move generation [8], pagando il prezzo di maggior memoria occupata. In sintesi: **DAWG** (file `.dawg`) offre compattezza e rapidità nelle lookup di parole, **GADDAG** (file `.gaddag`) offre massima velocità nella generazione di mosse consentendo di costruire parole da qualunque punto di aggancio (grazie ai prefissi invertiti) [9] [10]. Entrambi i formati sono supportati da Quackle, ma per il gioco in tempo reale il GADDAG è preferibile.

Va notato che dalla versione 1.0 Quackle introduce formati *versionati* per DAWG e GADDAG, con un controllo di consistenza via **hash MD5** incorporato [11] [12]. In particolare, ogni file .dawg include un MD5 del wordlist sorgente, e ogni .gaddag include un MD5 per verificare che sia **coerente con il DAWG** (ovvero che provenga dallo stesso insieme di parole) [11] [12]. Questo garantisce che il motore non carichi accidentalmente un GADDAG non corrispondente al dizionario atteso.

2. Build containerizzato e toolchain di compilazione

Compilare solo il core engine (headless): Per utilizzare Quackle in modalità CLI/headless (senza GUI Qt), conviene compilare solo le librerie core e gli strumenti console. Nel contesto Docker, assicurati di avere installato un toolchain C++ e CMake, oltre alle librerie Qt **base** (tipicamente QtCore e QtXml) necessarie per `quackleio` e i tool – **non serve** invece includere QtWidgets o compilare l'app Quackle GUI. Dal repository GitHub di Quackle si può usare CMake per generare tutto: ad esempio:

```
# Esempio in Dockerfile
RUN git clone https://github.com/quackle/quackle.git /src/quackle \
 && cd /src/quackle \
 && cmake -B build -S . -DCMAKE_BUILD_TYPE=Release \
 && cmake --build build --target makegaddag makeminidawg \
 && cmake --install build --prefix /usr
```

Il comando sopra compila i target `makegaddag` e `makeminidawg` (oltre alle librerie libquackle e quackleio richieste come dipendenze) e installa i binari e i dati in `/usr`. In alternativa, se la CMake di Quackle non offre un flag per **escludere la GUI**, si può semplicemente ignorare il target `quackle` (l'eseguibile Qt) concentrandosi sui tool. Il README ufficiale conferma che `libquackle` è pensato per essere linkato a "qualsiasi interfaccia conveniente" senza Qt [1]. Dunque, puoi compilare libquackle.a e libquackleio.a, e poi linkarli nel tuo progetto C++ wrapper. Nel wrapper C++, includi solo gli header di Quackle core (come `game.h`, `board.h`, etc.) e **non** quelli di Qt/GUI, e linka contro `-lquackle` e `-lquackleio` (oltre a QtCore/QtXml).

Percorsi per gli alfabeti e dati: Durante la fase di build/packaging nel container, occorre collocare i file di dati (alfabeti, dizionari) in un percorso dove Quackle li cercherà a runtime. Per default Quackle utilizza una directory "data" applicativa (spesso `/usr/share/quackle/data` su installazioni Unix) [13]. Il wrapper C++ di esempio imposta `QUACKLE_APPDATA_DIR` a `/usr/share/quackle/data` [13], che sarà usato da `DataManager` per trovare file di alfabeti e lexicon. Quindi, assicurati di **copiare i file .quackle_alphabet** necessari nella sottocartella `alphabets/` di tale directory. Ad esempio, dopo l'installazione, potresti avere `/usr/share/quackle/data/alphabets/english.quackle_alphabet` già presente (se installi i dati di Quackle) e dovrai aggiungere eventualmente `italiano.quackle_alphabet` per la lingua italiana (vedi sezione 3). Questo allineamento di percorsi fa sì che sia i tool (`makeminidawg`/`makegaddag`) sia il loader C++ trovino gli alfabeti e li interpretino in modo consistente. In alternativa, puoi definire la variabile d'ambiente `QUACKLE_APPDATA_DIR` nel container per puntare a una directory custom contenente `data/alphabets` e i tuoi file – il motore Quackle utilizzerà quella al posto del percorso di default.

Generazione di un dizionario DAWG (.dawg): Una volta compilati i tool, puoi generare il file lessicale in formato DAWG a partire dalla tua wordlist (file di testo con una parola per riga). Il tool da usare è `makeminidawg`. Esempio di comando (all'interno del container Docker):

```
makeminidawg wordlist.txt output.dawg
```

Il programma leggerà la lista di parole e produrrà `output.dawg`. La denominazione "mini" DAWG indica che Quackle utilizza un DAWG **minimale** (compresso ottimamente) grazie ad algoritmi di minimizzazione degli stati. *Nota:* assicurati che la wordlist sia codificata correttamente (preferibilmente

UTF-8) e **ordinata lessicograficamente**; se non lo è, `makeminidawg` potrebbe ordinarla internamente ma è buona pratica fornirla già in ordine. In caso di input disordinato, c'è rischio di un DAWG non minimale. Se la tua wordlist è molto grande, il processo può richiedere tempo e memoria: monitora l'utilizzo e considera di aumentare le risorse del container se necessario.

Generazione di un dizionario GADDAG (.gaddag): Per costruire il GADDAG puoi procedere in due modi: (a) direttamente dalla wordlist, oppure (b) a partire dal DAWG generato. Quackle fornisce il tool `makegaddag` che consente l'approccio (a) leggendosi il file di parole e scrivendo il .gaddag. Esempio di utilizzo:

```
makegaddag wordlist.txt output.gaddag
```

Il tool accetta come argomenti il percorso del file di input e quello di output (non legge da stdin – non usare redirection tipo `<<<`) [14]. Internamente, `makegaddag` utilizza la classe `GaddagFactory` di QuackleIO: questa accumula le parole, le **ordina** e genera la struttura GADDAG in memoria, per poi salvarla su file [15] [16]. Il processo è intensivo (il GADDAG di un dizionario di ~200k parole può occupare diverse decine di MB e impiegare vari secondi per generarsi). Durante la generazione vedrai stampe sul progresso (ogni 100k parole inserite) e al termine un log con la dimensione finale in byte del file `.gaddag` [17] [18].

L'approccio (b) – meno comune – consisterebbe nel creare prima il DAWG e poi "gaddag-izzare" il DAWG. In passato esistevano script o funzioni (`dawgfactory` + `gaddagize`) per convertire un DAWG in GADDAG. Tuttavia, con Quackle 1.0.x non è strettamente necessario, dato che puoi generare il GADDAG direttamente. Se scegli comunque di creare entrambi i formati, **assicurati che siano coerenti**: rigenera sempre il .gaddag dallo **stesso identico wordlist** usato per il .dawg, così gli hash MD5 interni coincideranno. In genere, se usi `makeminidawg` e `makegaddag` sul medesimo wordlist.txt, otterrai coppie .dawg/.gaddag compatibili (il GADDAG includerà l'MD5 del lexicon che corrisponde a quello presente nel DAWG) [11] [12].

Linkare il wrapper C++ con libquackle: Dopo aver generato i file .dawg/.gaddag, puoi procedere a integrare Quackle nel tuo wrapper C++ (ad esempio, per un servizio di gioco). Il wrapper deve linkare la libreria core (libquackle) e dovrebbe chiamare le API appropriate per caricare il lessico. Tipicamente si fa: inizializzare il **DataManager** singleton, impostare i parametri di gioco (alfabeto, regole, ecc.), quindi invocare `LexiconParameters::loadGaddag(path)` per caricare il file .gaddag in memoria [19] [20]. Nel tuo Dockerfile, oltre a copiare i file .gaddag e .dawg generati (es. sotto `/app/lexica/`), compila il wrapper e assicurati che all'esecuzione trovi le librerie e i dati. Puoi statically linkare libquackle.a nel tuo eseguibile wrapper per semplificare la distribuzione (nessuna dipendenza runtime, a parte le Qt base).

Riassumendo la toolchain DevOps: - **Build**: compila Quackle core + tools in container (no GUI). - **Generate lexicon**: con i tool, all'interno dello stesso container/immagine (per evitare mismatch di versione). - **Install data**: copia alphabet e lexicon files in `/usr/share/quackle/data` o percorso configurato. - **Compile wrapper**: linka contro libquackle, includi headers Quackle necessari. - **Runtime**: imposta `QUACKLE_APPDATA_DIR` se usi un path custom, altrimenti assicurati che i file siano nel path di default. Carica il .gaddag e verifica.

3. Alfabeti e internazionalizzazione dei dizionari

Formato di un file alfabeto `.quackle_alphabet`**:** Questi file definiscono l'insieme di lettere utilizzate in una data lingua, con i relativi punteggi e quantità di tessere. Un file `.quackle_alphabet`

è tipicamente un file **binario** serializzato da Quackle (non semplicemente testo leggibile). Viene prodotto quando si crea un nuovo lexicon tramite l'interfaccia Quackle, ma per i nostri scopi possiamo crearne uno usando il codice di Quackle o estendendo il core. In generale, l'alfabeto in Quackle è rappresentato da una serie di **LetterParameter** per ciascun simbolo: ogni lettera ha un **carattere** (es. "A"), un eventuale **carattere di "blank"** associato (es. "a" minuscola, usata per rappresentare quando un jolly vale quella lettera), un **punteggio** in punti e un **conteggio** di quante tessere di quella lettera sono presenti nel gioco [21] [22]. Inoltre c'è un flag isVowel (vocale) usato per valutazioni statistiche, ma è meno critico. Nel codice Quackle per l'alfabeto inglese ad esempio, la lettera 'A' ha punteggio 1 e conteggio 9, 'B' ha 3 punti e 2 tessere, … 'Z' ha 10 punti e 1 tessera [23] [24]. Questi parametri corrispondono esattamente alle regole di Scrabble per quella lingua. Oltre alle lettere alfabetiche, Quackle definisce anche alcuni marcatori speciali nel set: il **blank** (jolly, rappresentato internamente come `?`), e altri marcatori interni (come segnaposto per "played through" ecc.) [25]. Nel file alphabet questi compaiono come record per `?` con punteggio 0 e conteggio 2 (due jolly).

In pratica, per creare un nuovo file `.quackle_alphabet` per una lingua, il modo più diretto è scrivere un piccolo programma usando le API di Quackle: puoi creare un oggetto `AlphabetParameters`, utilizzare `setLetterParameter()` per ogni lettera con i valori appropriati, quindi serializzare l'oggetto su file. In alternativa, si può usare il formato testuale di config (non documentato ufficialmente) o modificare un file esistente. Un metodo empirico: prendi ad esempio `english.quackle_alphabet` già fornito e usalo come base sostituendo lettere/punteggi. **Tuttavia**, il formato non è immediatamente editabile a mano – più semplice è generarlo via codice oppure definire una classe custom nel core. Ad esempio, per l'italiano potresti copiare la classe `EnglishAlphabetParameters` (che nel costruttore inserisce tutti i LetterParameter per A-Z) e adattarla ai valori italiani [21] [24]. Poi istanzi questa classe nel DataManager.

Alfabeto italiano – struttura e differenze: L'italiano utilizza 21 lettere base (A-Z senza J, K, W, X, Y) più i due jolly. Inoltre le vocali accentate (À, È, É, Ì, Ò, Ù) in Scrabble **non vengono distinte**: qualsiasi parola con accento si gioca senza segni diacritici. Come conferma la distribuzione ufficiale, "**i segni diacritici sono ignorati**" [26]. Dunque il nostro alphabet per l'italiano conterrà solo le lettere non accentate. Anche le lettere straniere J, K, W, X, Y **non hanno tessere dedicate** in italiano (non esistono nel set standard) [27]. Ciò nonostante, parole con queste lettere possono apparire nel dizionario (prestiti linguistici): la regola è che se il giocatore vuole usarle deve impiegare un jolly. Per supportare questo scenario, conviene includere **anche J, K, W, X, Y nell'alfabeto ma con conteggio 0** [27]. In questo modo Quackle saprà riconoscere quelle lettere nelle parole del lexicon, ma non le metterà mai nel sacchetto (zero tessere). Il punteggio assegnato a queste lettere può essere impostato ai valori convenzionali (ad esempio 8 o 10 punti ciascuna, benché di fatto il loro punteggio non entri mai direttamente in gioco perché vengono coperte da un blank che vale 0).

Le distribuzioni e punteggi ufficiali per lo Scrabble italiano sono: 120 tessere totali, con 2 blank, **A**×14 (1 punto), **E**×11 (1 punto), **I**×12 (1 punto), **O**×15 (1 punto), **U**×5 (3 punti); **C**×6 (2 punti), **R**×6 (2 punti), **S**×6 (2 punti), **T**×6 (2 punti); **L**×5 (3 punti), **M**×5 (3 punti), **N**×5 (3 punti); **B**×3 (5 punti), **D**×3 (5 punti), **F**×3 (5 punti), **P**×3 (5 punti), **V**×3 (5 punti); **G**×2 (8 punti), **H**×2 (8 punti), **Z**×2 (8 punti); **Q**×1 (10 punti) [28] [29]. (Notare: la distribuzione italiana assegna 8 punti a G, H, Z e 10 a Q). Le lettere J, K, W, X, Y come detto non sono presenti (le considereremo a conteggio 0).

Per creare `italiano.quackle_alphabet`, puoi procedere così: definisci i LetterParameter per ogni lettera dell'alfabeto italiano esteso (A–Z) in ordine. Per A, punteggio=1, conteggio=14, vocale=true; B=5,3,false; … via così secondo i dati sopra. Imposta J,K,W,X,Y con count=0 e un score (puoi assegnare 8 o 10 in base alla difficoltà prevista, ad es. J=8, K=5, W=8, X=8, Y=8 come reference, o lasciare 0 – ininfluente). Due blank con count 2 sono automaticamente gestiti (nel costruttore

`AlphabetParameters` base vengono creati i simboli speciali `?` per blank)[25], ma occorre aggiornare il conteggio jolly a 2 esplicitamente: e.g. `setCount(QUACKLE_BLANK_MARK, 2)`. Una volta assemblato l'alfabeto, salvalo su file .quackle_alphabet.

Integrazione dell'alfabeto nei tool e nel loader: Affinché i tool di generazione e il motore usino l'alfabeto corretto, occorre indicare loro la scelta della lingua. Il modo più semplice è nominare il file appropriatamente e usarlo al posto di quello inglese. Ad esempio, se chiami il file `italian.quackle_alphabet` e lo metti in `data/alphabets/`, potrai istruire Quackle a usarlo. Nel caso dell'interfaccia grafica, l'utente sceglierebbe "italian" come alphabet. Nel nostro contesto headless, dobbiamo farlo programmaticamente. Nel wrapper C++ prima di caricare il GADDAG conviene chiamare:

```
QUACKLE_DATAMANAGER->setAlphabetParameters(new
Quackle::EnglishAlphabetParameters());
```

Nel codice attuale d'esempio, effettivamente viene forzato l'alfabeto inglese di default[30]. Dovrai sostituire quella riga creando un'istanza della tua classe ItalianAlphabetParameters (se l'hai implementata in core) oppure caricando il file: sfortunatamente Quackle non espone direttamente un metodo "load alphabet file" semplice, ma potresti simulare l'operazione caricando un lexicon con quell'alfabeto. Un trucchetto: puoi sfruttare `FlexibleAlphabetParameters` di QuackleIO, che adatta l'alfabeto in base all'input. Nel tool `makegaddag` vediamo che viene creato un `FlexibleAlphabetParameters flex` vuoto[31] prima di inserire le parole; ciò fa sì che GaddagFactory usi un alfabeto flessibile (in pratica assume l'alfabeto di default se non diversamente specificato). Per sicurezza, conviene *comunque* che il wordlist non contenga caratteri al di fuori di A-Z, oppure che tu rimuova accenti e caratteri speciali in precedenza (vedi sotto). In definitiva, l'ordine delle operazioni sarà: impostare l'alfabeto (Italiano) nel DataManager, poi chiamare `LexiconParameters::loadGaddag("italiano.gaddag")`. Quackle verificherà la lunghezza dell'alfabeto atteso nel file .gaddag rispetto a quella corrente e l'**MD5** del lexicon. Se l'alfabeto non corrisponde (es. diverso numero di lettere) potrebbe segnalare errore o persino generare un fault, quindi è fondamentale che l'alfabeto impostato sia quello corretto con cui il GADDAG è stato generato.

Normalizzazione e pulizia delle wordlist: Prima di generare i file lessicali, assicurati di pulire la lista parole in base alle regole della lingua Scrabble. Per l'italiano, come detto, **rimuovi gli accenti** dalle parole (e altre diacritiche come eventuali apostrofi o spazi). Ad esempio, "**città**" diventa "**citta**", "**perché**" -> "**perche**". Puoi usare strumenti come Python (unicodedata normalize) o `iconv`: ad esempio `iconv -f UTF-8 -t ASCII//TRANSLIT` su una parola con accenti spesso restituisce la versione senza accento. Rimuovi anche eventuali **apostrofi o altri caratteri non alfabetici**: il lexicon di Quackle deve contenere solo lettere A-Z. Parole composte o con apostrofo (es. "guarda-là" o "l'altro") vanno spezzate o eliminate secondo le convenzioni (di solito, lo Scrabble ammette solo parole singole senza spazi/punti). Converte tutto in **maiuscolo** prima di passare ai tool – `makegaddag` lo fa già internamente (usa `std::toupper` su ogni riga)[32], ma conviene già fornirgli input uppercase per evitare sorprese con caratteri speciali. Infine, **deduplica e ordina** la lista: elimina duplicati e assicurati che sia in ordine alfabetico crescente (A->Z). Strumenti come `sort` e `uniq` su Unix possono aiutare. Questa "pipeline" di normalizzazione garantisce che il DAWG/GADDAG risultante sia corretto e non contenga voci spurie. Anche la corrispondenza con l'alfabeto sarà assicurata: dopo la pulizia, tutte le lettere presenti nel wordlist dovrebbero appartenere all'alfabeto definito (nessun carattere fuori insieme). Un controllo manuale utile: estrai l'insieme di caratteri usati nel wordlist (ad es. con grep o uno script Python) e verificalo rispetto all'alfabeto atteso.

4. Errori comuni, debugging e soluzioni

L'integrazione di Quackle e la generazione dei lexicon non è banale; di seguito alcune problematiche frequenti e come affrontarle:

- `lexicon_ok: false` **nel wrapper** – Questo indica che il motore non è riuscito a caricare correttamente il dizionario .gaddag. Nel wrapper C++ di esempio, dopo il tentativo di `loadGaddag` viene stampato un JSON con `"lexicon_ok": false` e un errore [33]. Le cause comuni: file .gaddag mancante o percorso errato, formato non valido (corrotto), oppure **mismatch di versione**. In caso di version mismatch, Quackle 1.0.4 rileva che l'header del file non corrisponde a quello atteso e lancia un'eccezione. Nel wrapper ciò genera un messaggio di errore *"invalid or unsupported GADDAG file (wrong version/corrupt). Regenerate with the same Quackle revision as libquackle."* [34]. La soluzione è rigenerare il file .gaddag usando **la stessa versione** di Quackle del motore in uso (idealmente, generarlo dentro lo stesso container in cui gira il motore, come da best practice DevOps). Se l'errore è "file not found", ovviamente controlla i path: puoi settare un log su stderr nel wrapper (come già fa stampando `[wrapper] loading gaddag path=...` all'avvio [35]) per confermare il percorso.

- **Segmentation fault all'avvio (crash nel load)** – Un crash improvviso durante `loadGaddag` (senza messaggi di eccezione) di solito indica un problema più grave di compatibilità. Ad esempio, se si tenta di caricare con Quackle 0.97 un file .gaddag creato con Quackle 1.0.x, le differenze di formato (e la mancanza di controlli di versione nelle vecchie versioni) possono causare letture errate in memoria e crash. Analogamente, un alphabet inconsistente può portare il motore a indicizzare fuori dai limiti interni. Dal 2015 in poi, con i formati versionati, questi casi dovrebbero essere gestiti con errori graziati (es. eccezioni), ma se utilizzi versioni miste potresti incorrere in segfault. **Verifica sempre la "matrice di compatibilità":** Quackle 1.0.4 può leggere file generati da 1.0.0–1.0.4 (stessa major), ma non leggerà quelli di 0.9x; viceversa, una versione pre-1.0 non saprebbe interpretare i file nuovi. La regola è: **stessa versione per build engine e build lexicon**.

- **Formato non valido / MD5 mismatch** – Può capitare che un file .dawg o .gaddag venga rigettato perché "lexicon hash mismatch". Ad esempio, se rigeneri il GADDAG da una wordlist diversa ma mantieni il nome di un dizionario esistente, Quackle rileverà che l'hash MD5 del contenuto non corrisponde a quello atteso (magari perché il .dawg abbinato è differente). In UI verrebbe indicato come errore di dizionario; nel nostro contesto headless potresti vedere solo `lexicon_ok=false`. Per approfondire, puoi ricompilare il wrapper con logging più dettagliato o eseguire il tool `--check-gaddag` fornito (vedi sotto) per un responso semplice. La soluzione è assicurarsi di usare i file abbinati giusti. Se hai sia .dawg che .gaddag, puoi anche provare a caricare il .dawg al posto del .gaddag (Quackle dovrebbe saper caricare anche direttamente il DAWG); se uno dei due file è obsoleto o sbagliato, almeno il .dawg potrebbe dare un errore MD5 più esplicito.

- **File di alphabet non trovato o errato** – Se il DataManager non trova il file .quackle_alphabet nominato nel lexicon, potrebbe usare l'alfabeto di default ("english") senza avvisare, causando però discrepanze. Ad esempio, se carichi un dizionario italiano ma senza aver impostato l'alfabeto italiano, il motore userà l'inglese: potresti notare che alcune lettere non vengono riconosciute e mosse valide mancano. Un sintomo è che il motore genera segnalazioni come lettere non valide o punteggi errati. Per evitare ciò, assicurati di **impostare l'alfabeto corretto prima di loadGaddag**. Nel wrapper, dopo il `setAlphabetParameters(...)` puoi verificare che `QUACKLE_DATAMANAGER->alphabetParameters()->length()` corrisponda al numero

di lettere atteso (21 per italiano vs 26 per inglese). Se hai dubbi che stia caricando l'alfabeto giusto, puoi forzare manualmente la lettura del file: ad esempio, usando `DataManager::findDataFile("alphabets", name)` [36] per verificare il path, e magari aggiungere un log quando viene aperto.

- **Errori di shell e pipeline** – Quando si integrano i tool in script shell o Docker, fai attenzione all'uso corretto della redirezione. Come accennato, `makegaddag` e `makeminidawg` leggono da **file** passati come argomento, non supportano input via STDIN pipe a meno di usare espedienti (p.es. `makeminidawg /dev/stdin out.dawg` può funzionare, ma è preferibile un file temporaneo). Inoltre, la redirezione *here-string* `<<<` produce un EOF immediato dopo una sola riga, quindi non va bene per passare un intero file di migliaia di parole. In uno script Bash conviene usare: `makeminidawg "$wordlist" "$output"` normalmente, oppure `cat words.txt | makeminidawg /dev/stdin out.dawg` se necessario. Se dimentichi gli argomenti, i tool stamperanno l'uso corretto e termineranno (es. *Usage: makegaddag <wordlist.txt> <out.gaddag>* [14]).

Strategie di debugging: Per diagnosticare problemi, sfrutta il più possibile il logging già presente. Il wrapper, come visto, stampa su stderr messaggi con prefisso `[wrapper]`. In caso di crash silenziosi, puoi eseguire il wrapper sotto `gdb` nel container o abilitare core dump. Se sospetti un problema nel file .gaddag, disponi di un comodo comando integrato: eseguendo `engine_wrapper --check-gaddag path/to/file.gaddag` (come implementato nel wrapper FastAPI) il processo tenterà solo di caricare il file e terminare [37] [19]. Il codice restituisce 0 se il gaddag è ok, oppure un codice di errore e un messaggio specifico se fallisce (file non trovato, impossibile aprire, file vuoto, eccezione durante load...). Questa modalità è utile da inserire magari nella fase di avvio: lo script FastAPI fornito, ad esempio, esegue un controllo all'startup e logga un messaggio FATAL se il check fallisce, consigliando di rigenerare il file con la stessa versione di Quackle [38]. Anche tu puoi implementare qualcosa di simile come **self-test**: dopo aver generato i file, fai girare `--check-gaddag` per assicurarti che quell'eseguibile li digerisca correttamente (così eviti di scoprirlo solo a runtime avanzato).

Un'altra tecnica di debug è attivare eventualmente le **assert** in Quackle: se compili in Debug, Quackle ha vari assert interni (es. sulla costruzione dell'alfabeto, sulle lookup di lettere) che possono segnalare incongruenze subito [39] [40]. In produzione userai Release, ma durante lo sviluppo se incontri crash inspiegabili, ricompilare Quackle con simboli di debug e assert può dare indicazioni.

Risoluzione dei mismatch alphabet/lexicon: Ricapitolando, se scopri un mismatch tra alfabeto e lexicon (ad es. hai modificato il wordlist includendo lettere prima assenti), è necessario **rigenerare** il dizionario con l'alfabeto aggiornato. Ogni volta che cambi qualcosa nel set di lettere o nel wordlist, invalidi il .gaddag precedente. Mantieni quindi un processo di build idempotente: conviene scrivere uno script (o Dockerfile step) che partendo dalla stessa wordlist e alphabet generi sempre i .dawg/.gaddag. In questo modo, se fai modifiche, rigeneri semplicemente ricostruendo l'immagine. Per evitare confusioni, puoi includere nel nome del file o nella versione API un identificatore (es. versione dizionario). Quackle 1.0.4 internamente risolve in parte questo problema con l'MD5, ma a livello di deployment, ad esempio, potresti calcolare un hash del file .gaddag e confrontarlo su startup per vedere se corrisponde a quello atteso.

Attenzione anche a **locale/encoding** nel container: se la wordlist contiene caratteri non ASCII e la locale di sistema non è UTF-8, la lettura potrebbe fallire o alterare i caratteri. Imposta `LANG=C.UTF-8` o simili nell'ambiente. Tuttavia, come detto, è meglio evitare del tutto caratteri speciali nel wordlist (già normalizzati).

5. Best practices e risorse utili

Per lavorare con Quackle efficacemente, tieni presenti alcune best practice e risorse della community:

- **Usa l'ultima versione stabile (1.0.4)**: Dato che Quackle 1.0 ha introdotto formati robusti con hash e versioning, si raccomanda di usare la serie 1.0.x per i progetti odierni. La 1.0.4 (luglio 2019) include tutte le patch di stabilità. In particolare, versioni 1.0.x garantiscono che GADDAG e DAWG contengano un MD5 di verifica [11], evitando molti bug strani di incoerenza che affliggevano le vecchie versioni. Ad esempio, prima di 1.0 era noto che caricando un .bin (vecchio formato lexicon) sbagliato Quackle poteva crashare senza spiegazioni. Ora invece fallisce con errore controllato se l'hash non quadra. Leggi il changelog sul sito Quackle (sezione "New Features in Quackle 1.0") per i dettagli [11] [12].

- **Genera i dizionari in** locale **con la stessa toolchain**: Non creare il .gaddag a mano o su un sistema diverso da quello dove gira l'engine. Idealmente, includi la generazione nel Docker build, così sei certo che `libquackle` e i file lessicali provengano dallo stesso codice. Questo elimina ogni rischio di incompatibilità. Inoltre, rigenera sempre sia .dawg che .gaddag assieme (anche se magari usi solo il .gaddag): avere entrambi non guasta, e il .dawg può servire per debug o verifiche (oltre a occupare molto meno spazio, utile per controlli manuali con tool esterni se mai servisse).

- **Mantieni allineati nome lexicon e file**: Quando carichi un dizionario con Quackle, tipicamente gli associ un nome (es. "Italiano") e il motore si aspetta di trovare un .dawg/.gaddag corrispondente a quel nome, oltre che l'alfabeto relativo. Nel wrapper, ad esempio, si chiama `setBackupLexicon("enable1")` [41] – questo indica il lexicon di default. Se stai creando un dizionario italiano, potresti seguire lo schema e chiamarlo "italiano" in analogia. Non è strettamente necessario, ma nominare coerentemente i file (italiano.quackle_alphabet, italiano.dawg, italiano.gaddag) riduce confusione. Potresti anche voler modificare `setBackupLexicon("italiano")` nel DataManager, così se in futuro integri l'AI completa, saprà qual è il dizionario di fallback.

- **Contribuisci risorse alla community**: Ci sono state discussioni preziose nel gruppo Yahoo "Quackle" (ora archiviato) e su forum come Reddit r/scrabble, riguardo a creazione di dizionari personalizzati. Ad esempio, utenti hanno chiesto come modificare il **tile bag** (set di lettere) e la risposta tipica è di creare un file .quackle_alphabet personalizzato e metterlo in `.../alphabets/` [26]. Anche se quei thread possono essere difficili da recuperare oggi, vale la pena cercare su Internet parole chiave come "Quackle custom dictionary GADDAG" per trovare eventuali guide o script di altri. Inoltre, il codice sorgente stesso di Quackle è un'ottima documentazione: file come `alphabetparameters.cpp` mostrano come è costruito l'alfabeto inglese in codice [21] [24], e `gaddagfactory.cpp` (in QuackleIO) spiega come vengono scritti i file GADDAG.

- **Letture consigliate su GADDAG e DAWG:** Per comprendere a fondo perché il GADDAG è importante, puoi leggere l'articolo originale di Steven A. Gordon, *"A faster Scrabble move generation algorithm"* (Software: Practice & Experience, 1994). È lì che fu presentata la struttura GADDAG e si dimostra il guadagno di velocità ~2x sul DAWG [7]. Un riassunto di quell'articolo è anche presente sulla pagina Wikipedia del GADDAG [42] [43], che abbiamo citato. In breve, il GADDAG permette di generare parole partendo da ogni possibile lettera di ancoraggio già sul tabellone, esplorando sia a sinistra (prefisso invertito) che a destra (suffisso) in un unico percorso nel grafo [9] [10]. Per chi implementa motori di Scrabble, questa è una lettura fondamentale.

Anche la tesi di Nick Meller (*"Algorithms in Scrabble"*, 2017) e alcune discussioni sul forum of the Scrabble programming community possono dare idee su come ottimizzare ulteriormente (ad es. alternative come *Directors* e *Tries*).

- **Performance e memory footprint:** Tieni conto che un GADDAG occupa molta RAM (anche 5× un DAWG) [7]. In produzione Docker su server con risorse limitate, monitora l'uso memoria. Se stai usando ad esempio il dizionario inglese ENABLE (178k parole circa), il .gaddag può pesare ~15-20 MB e in memoria ancora di più. Il caricamento iniziale potrebbe richiedere qualche centinaio di ms. Se questo è critico, potresti valutare di caricare il dizionario una volta sola e riutilizzare il processo (come fa il wrapper FastAPI con un processo engine persistente).

In ultimo, vale la pena sfogliare il repository Quackle per i file `FAQ.md` o `README.md` aggiuntivi: spesso contengono note sulle lingue supportate (ad esempio, Quackle 0.98 menzionava il supporto ad alfabeti con >32 lettere [44], aggiungendo Slovacco – segno che ormai il formato è generalizzato). Ogni nuova lingua potrebbe avere piccole insidie (es. per il francese bisogna caricare anche la lista di *Collins* abbreviata, ecc.), ma l'impianto rimane lo stesso.

6. Integrazione backend: wrapper C++ e API FastAPI

Infine, diamo uno sguardo a come utilizzare il motore Quackle generato all'interno di un servizio, ad esempio con un'API FastAPI come nel tuo scenario:

Wrapper C++ con I/O JSON: Il wrapper `engine_wrapper` è un programma C++ (basato su Quackle core) che legge comandi in formato JSON da **STDIN** e stampa risultati in JSON su **STDOUT**. Questo design consente di eseguire il motore come subprocess e comunicare con esso tramite pipe dal server Python. All'avvio, puoi passare parametri come `--gaddag path` e `--ruleset it/en` per specificare il file di dizionario e la lingua [45] [46]. Il wrapper inizializza Quackle (crea DataManager, imposta la directory dati, i parametri di base inglesi, e poi carica il GADDAG) [13] [47]. Nel tuo caso, per supportare l'italiano, dovrai modificare quella fase di init (come detto sopra) per usare l'alfabeto italiano e magari un diverso board (Scrabble italiano usa lo stesso tabellone 15x15, quindi `EnglishBoard` va bene). Una volta caricato il lexicon, il wrapper entra in loop leggendo input. Supporta comandi come `"ping"` (cui risponde con `{"pong": true}` per test di vivacità) [48], `"probe_lexicon"` (per verificare se il dizionario è caricato correttamente e ottenere info come path e size) [49] [50], e soprattutto `"compute"` per calcolare le mosse migliori. Un esempio di comando JSON che il server invia potrebbe essere:

```
{
  "op": "compute",
  "board": [ ["", " ", " ", ...], ""], ... 15 righe ... ],
  "rack": "NERAIS?",
  "limit_ms": 1500,
  "top_n": 10
}
```

Qui `"board"` è una matrice 15x15 con stringhe di 1 carattere (lettere già sul tavolo, oppure vuota/space per caselle vuote), `"rack"` sono le lettere sulla rastrelliera del giocatore (es. 7 lettere, `?` indica un jolly), `limit_ms` un timeout in millisecondi per limitare la ricerca, `top_n` quante mosse migliori si vogliono. Il wrapper valida l'input, prepara lo stato di gioco: crea i giocatori, inizializza il Board vuoto, imposta il rack corrente [51] [52], piazza sul board tutte le lettere fisse dal JSON (tramite

`board.makeMove` per ogni lettera esistente) [53] [54]. Poi invoca il **Generator** di Quackle: `Generator gen(pos); gen.allCrosses(); gen.kibitz(n);` – questo istruisce Quackle a generare tutte le mosse legali e calcolare i punteggi, restituendo la lista ordinata delle migliori [55] [56]. L'output viene raccolto e serializzato in JSON: per ogni mossa include la parola formata, la posizione (riga, colonna, direzione) e il punteggio [57]. Ad esempio, una risposta tipica potrebbe essere:

```
{
  "moves": [
    {"word": "SIRENA", "row": 7, "col": 8, "dir": "H", "score": 36,
"positions": [[7,8],[7,9],[7,10],[7,11],[7,12],[7,13]]},
    {"word": "RASINE", "row": 6, "col": 7, "dir": "V", "score": 28,
"positions": [[6,7],[7,7],[8,7],[9,7],[10,7],[11,7]]},
    ...
  ],
  "truncated": false
}
```

In cui `moves` è una lista delle migliori mosse trovate (ordinate per score decrescente), e ogni mossa specifica la parola, coordinate iniziali, direzione (`"H"` orizzontale, `"V"` verticale), punteggio calcolato e le coordinate di ogni lettera posizionata sulla board (per comodità del frontend). Il campo `"truncated"` è false se la ricerca ha concluso entro il tempo; se il calcolo eccede `limit_ms`, il wrapper interrompe l'elaborazione e risponde con `truncated: true` (in questa implementazione semplice, se scade il tempo restituisce una lista vuota [58], ma volendo si potrebbe mantenere le mosse trovate fin lì).

Endpoint FastAPI `/engine/cmd`**:** Sul lato Python/FastAPI, si avvia il processo wrapper come descritto nel file `main.py`. Il codice prevede una funzione `start_engine()` che lancia il subprocess `engine_wrapper` con i parametri opportuni [59] [60] e avvia un thread per leggere continuamente lo stderr del processo (log del wrapper) così da inoltrarlo all'output del server per debugging [61] [62]. L'endpoint `/engine/cmd` è definito per accettare qualsiasi payload JSON e lo inoltra alla funzione `ask_engine(payload, timeout)` [63]. Quest'ultima scrive il JSON sulla `stdin` del processo (*engine_proc.stdin.write*) [64] e attende una linea di risposta su `_engine_proc.stdout` [65] [66]. In caso di timeout o errore, viene sollevata eccezione HTTP 504 o 500. In caso di successo, restituisce direttamente il JSON parsato dal wrapper. Questo significa che dall'esterno (es. un client web) puoi effettuare richieste POST a `/engine/cmd` con body JSON come quello mostrato sopra, e il server risponderà con il JSON delle mosse. In aggiunta, il codice espone endpoint dedicati come `/api/v1/move` che incapsula e valida una richiesta di mossa con schema Pydantic (MoveRequest/MoveResponse) – questo è più user-friendly e potrebbe trasformare l'output in un formato più adatto all'applicazione. Ma concettualmente, il `/engine/cmd` è universale e ti permette anche di inviare comandi come `{"op":"ping"}` o `{"op":"probe_lexicon"}` e ottenere le risposte di servizio dal wrapper. Ad esempio, `/health/lexicon` chiama `probe_lexicon` e risponde con `lexicon_ok` e info di hash se disponibili [67] [68].

Logging e monitoraggio: Nel contesto container, è importante che i log del wrapper (stderr) vengano catturati. Il thread `_drain_stderr` fa proprio questo: stampa ogni linea di stderr con prefisso `[wrapper]` sul log del server Python [69], così potrai vedere nel log container messaggi come `[wrapper] loading gaddag path=...` o eventuali errori runtime del motore. Questo aiuta molto a capire cosa succede senza dover attaccare un debugger al processo C++. Assicurati di mantenere questa funzionalità o di integrare un sistema di log simile.

Consistenza stato engine: Poiché il wrapper process rimane vivo tra richieste, esso mantiene in memoria il board e il rack dell'ultima richiesta elaborata. Nel codice fornito, ogni volta che arriva un nuovo compute, viene creato un nuovo `GamePosition` e popolato da zero, quindi lo stato precedente viene sovrascritto e non persiste – il che è corretto per questo scenario (ogni richiesta è indipendente) [70] [53]. Se volessi gestire una partita mossa dopo mossa con stato persistente, dovresti cambiare approccio (ad es. mantenere `Game` e applicare le Move di volta in volta). Ma per un suggeritore di mosse singole va bene così. Ricorda solo che se per qualunque ragione il processo engine si interrompesse (crash o exit), il server rileverà `_engine_proc.poll() != None` e tenterà di riavviarlo alla prossima richiesta [71]. Ciò aggiunge robustezza: se per esempio un segfault sfuggisse e il processo morisse, il prossimo `ask_engine` proverà a `start_engine()` di nuovo. È buona norma quindi implementare la terminazione pulita: ad esempio, potresti aggiungere un handler per il segnale SIGTERM nel wrapper C++ che chiude DataManager e rilascia memoria, ma in pratica uscire dal processo equivale a farlo riavviare quindi va bene anche "kill and respawn" se gestito.

Endpoint di health: Nel file Python ci sono anche `/healthz`, `/health/lexicon`, `/health/engine` per controlli di stato [72] [73]. Questi sono utili in ambiente Kubernetes o simili, per verificare che il container sia sano. `/health/engine` usa il ping/pong per vedere se il wrapper risponde velocemente. `/health/lexicon` chiama `probe_lexicon` e riporta se il dizionario è caricato e coerente [67]. È un ottimo esempio di come puoi sfruttare i comandi del wrapper per introspezione.

In conclusione, questa integrazione dimostra un approccio efficiente: **process isolation** per l'engine nativo e comunicazione tramite JSON. Ciò ti permette di beneficiare della velocità del C++ per la generazione di mosse, mantenendo la comodità di orchestrazione e rete in Python.

Riepilogo dei passaggi chiave per generare e caricare un GADDAG valido:

1. **Prepara la wordlist** – Pulisci il file di parole (una per riga) rimuovendo caratteri non supportati, normalizzando accenti e maiuscole, eliminando duplicati. Verifica che contenga solo lettere dell'alfabeto target.

2. **Definisci l'alfabeto** – Crea il file `.quackle_alphabet` per la lingua (se non già esistente). Inserisci tutte le lettere con punteggi e quantità corretti, includendo eventuali lettere "esterne" con count 0. Posiziona questo file in `data/alphabets/` nel container (es. `/usr/share/quackle/data/alphabets/italiano.quackle_alphabet`).

3. **Compila Quackle e strumenti** – Nel tuo Docker, compila libquackle e i tool console. Assicurati di includere le Qt base per quackleio. (Se usi l'immagine base già pronta con Quackle, verifica che sia la versione 1.0.4 per coerenza).

4. **Genera il DAWG (opzionale)** – Esegui `makeminidawg wordlist.txt italiano.dawg`. Controlla l'output per assicurarti che abbia letto il numero giusto di parole e scritto il file .dawg senza errori.

5. **Genera il GADDAG** – Esegui `makegaddag wordlist.txt italiano.gaddag`. Verifica nel log la dimensione e che il processo termini con successo. Per ulteriore sicurezza, puoi usare `engine_wrapper --check-gaddag italiano.gaddag` subito dopo, che dovrebbe restituire exit code 0 se tutto è ok [38]. Questo comando controllerà anche l'MD5 interno: se per qualche motivo il .gaddag fosse incompatibile col .dawg generato, te ne accorgeresti qui.

6. **Distribuisci i file** – Copia `italiano.gaddag` (e volendo `italiano.dawg`) nella directory prevista nel container (es. `/app/lexica/` come da configurazione ENV). Accertati che il wrapper sappia dove cercare: nel nostro esempio, viene passato via `--gaddag /app/lexica/italiano.gaddag` all'avvio.

7. **Configura il wrapper per la lingua** – Modifica/parametrizza il wrapper C++ in modo che imposti l'alfabeto italiano. Se hai un flag `--ruleset it`, usalo per scegliere l'alfabeto. Puoi implementare ad esempio: `if (cfg.ruleset == "it") QUACKLE_DATAMANAGER->setAlphabetParameters(new ItalianAlphabetParameters());` prima di loadGaddag. In mancanza di ciò, caricherà di default l'inglese [30] con risultati errati, quindi questo passo è cruciale.

8. **Carica il GADDAG nel motore** – Avvia il wrapper e monitora l'output: dovresti vedere `[wrapper] loading gaddag path=...` e poi eventualmente un log di successo (potrebbe stampare qualcosa tipo `gaddag-ok size=N bytes`). Se c'è un problema, apparirà subito un errore su stderr (ad esempio file non trovato o eccezione formato).

9. **Testa con query di esempio** – Usa l'endpoint `/engine/cmd` (o equivalente CLI se non hai server) per inviare un comando di ping (`{"op":"ping"}`) e assicurarti che risponda con pong. Poi prova una richiesta di calcolo con una situazione semplice (anche board vuoto e rack di 7 lettere) per vedere se restituisce mosse sensate. Controlla `lexicon_ok` via `/health/lexicon` se disponibile, per conferma finale [68].

Seguendo questi passaggi, otterrai un file GADDAG coerente e caricato correttamente nel wrapper C++, eliminando gli errori di incompatibilità. In caso di dubbi, ricorda di fare riferimento alle fonti citate (documentazione Quackle, discussioni online) e non esitare a sperimentare in ambiente di test. Con questo "deep dive", dovresti avere gli strumenti per integrare Quackle 1.0.4 con dizionari personalizzati (come l'italiano) in modo containerizzato, robusto e performante. Buon coding e... buone parole! [7] [12]

---

[1] [2] [3] [4] GitHub - quackle/quackle: Quackle crossword game artificial intelligence and analysis tool
https://github.com/quackle/quackle

[5] [6] [7] [8] [9] [10] [42] [43] GADDAG - Wikipedia
https://en.wikipedia.org/wiki/GADDAG

[11] [12] [44] Quackle - kwak!
https://people.csail.mit.edu/jasonkb/quackle/

[13] [19] [20] [30] [33] [34] [35] [37] [41] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [70] engine.cpp
https://github.com/Jacopo888/scarabeo-ace-44/blob/cf5f3546252c5de717589fb13ff015b977c493ae/engine/quackle_wrapper/engine.cpp

[14] [15] [16] [17] [18] [31] [32] main.cpp
https://github.com/Jacopo888/scarabeo-ace-44/blob/cf5f3546252c5de717589fb13ff015b977c493ae/engine/tools/makegaddag/main.cpp

[21] [22] [23] [24] [25] [36] [39] [40] alphabetparameters.cpp
https://github.com/quackle/quackle/blob/c68ac871c73ff4c76246fc49465f53f8befef92f/alphabetparameters.cpp

[26] [27] [28] [29] Scrabble letter distributions - Wikipedia
https://en.wikipedia.org/wiki/Scrabble_letter_distributions

38 59 60 61 62 63 64 65 66 67 68 69 71 72 73 main.py

https://github.com/Jacopo888/scarabeo-ace-44/blob/cf5f3546252c5de717589fb13ff015b977c493ae/engine/app/main.py