1: `solve(self, current_state)` .
- این متد یک راه حل برای بازی "Water Sort" پیدا می‌کند، اما لزوماً بهینه نیست.
- الگوریتم آن بر اساس جستجوی عمق‌اول (Depth-First Search) است.
- در هر مرحله، این متد تمام حرکات ممکن را در `actions(self, current_state)` پیدا می‌کند.
- سپس به صورت تکراری، هر کدام از این حرکات را روی حالت فعلی اعمال می‌کند و به یک حالت جدید می‌رسد.
- اگر حالت جدید به پیروزی منجر شود، `self.solution_found`را به `True` تغییر داده و راه حل را در `self.moves` ذخیره می‌کند.
- در غیر این صورت، به طور بازگشتی متد `solve()` را روی حالت جدید فراخوانی می‌کند.
- اگر هیچ راه حلی پیدا نشود و `self.moves` خالی باشد، آخرین حرکت از `self.moves`حذف می‌شود.

2: `optimal_solve(self, current_state)` .
- این متد یک راه حل بهینه (با کمترین تعداد حرکات) برای بازی "Water Sort" پیدا می‌کند.
- الگوریتم آن بر اساس جستجوی A* است.
- در هر مرحله، این متد یک گراف از حالات بازی ایجاد می‌کند.
- هر گره در این گراف شامل موارد زیر است:
`g`: - تعداد حرکات انجام شده تا به این حالت رسیده‌ایم
`state`: - حالت فعلی بازی
`move`: - حرکتی که منجر به این حالت شده است
`parent`: - گره والد، که از طریق آن به این حالت رسیده‌ایم
- برای هر حالت جدید، یک تخمین هیورستیک (`h(self, state)`) از فاصله آن تا حالت پیروزی محاسبه می‌شود.
- این گره‌ها در یک پایگاه داده اولویتی (Priority Queue) قرار داده می‌شوند، به طوری که گره‌هایی با کمترین `g + h` هزینه + هیورستیک) در اولویت بالاتری قرار گیرند.
- در هر مرحله، گره با کمترین `g + h` از پایگاه داده خارج شده و بررسی می‌شود.
- اگر این گره به پیروزی منجر شود، `self.solution_found`را به `True` تغییر داده و مسیر را در `self.moves` ذخیره می‌کند.

- در غیر این صورت، تمام حرکات ممکن از این گره را بررسی و گره‌های جدید را به پایگاه داده اضافه می‌کند.
- این فرایند تا زمانی ادامه می‌یابد که یک راه حل بهینه پیدا شود.

به طور کلی، `solve()` یک راه حل ساده‌تر اما ممکن است بهینه نباشد، در حالی که `optimal_solve()` یک راه حل بهینه اما پیچیده‌تر پیدا می‌کند.